Final Report for Period: 08/2011 - 07/2012
Submitted on: 09/19/2012
Principal Investigator: Prvulovic, Milos .
Award ID: 0903470
Organization: Georgia Tech Research Corp
Submitted By:
Prvulovic, Milos - Principal Investigator
Title:
Performance Debugging Support for Many-Core Processors

Project Participants

Senior Personnel

Name: Prvulovic, Milos
Worked for more than 160 Hours: Yes
Contribution to Project:

Post-doc

Graduate Student

Name: Oh, Jungju
Worked for more than 160 Hours: Yes
Contribution to Project:
Jungju is the main graduate student working on this project. His support comes about 1/2 from the NSF portion and 1/2 from the SRC portion of this project.

Name: Park, Sunjae
Worked for more than 160 Hours: Yes
Contribution to Project:

Undergraduate Student

Technician, Programmer

Other Participant

Research Experience for Undergraduates

Organizational Partners

Semiconductor Research Corporation
This NSF award is part of a NSF/SRC jointly funded project. As a result, SRC is funding this project at $30,000 per year.

Intel Corporation
We have weekly meetings with Dr. Christopher Hughes form Intel Research (Santa Clara, CA) in which we discuss the ideas and the progress on this project. Dr. Hughes is also serving as out industrial liaison for the SRC portion of this project.

Freescale Semiconductor, Inc, Technology Solutions Organization
We have had interactions with Roy Sourav from Freescale Semiconductor about extending our work on load imbalance debugging to embedded applications that are of interest to Freescale. We have gained valuable insights into the needs of the programmers in the embedded multi-core and many-core space.

Other Collaborators or Contacts

Dr. Christopher Hughes (Intel Research) is participation in this project through weekly teleconference meetings. These meetings are extremely valuable because Dr. Hughes is one of the top experts on emerging many-core applications and is very interested in performance debugging of such applications.

Dr. Guru Prasadh Venkataramani, who was a graduate student working on the preliminary exploration for this project, has graduated only days before the official starting date of this project and is now a faculty member at George Washington University. Because he was involved in our preliminary work, Dr. Venkataramani continued participating in our weekly teleconference meetings during the first 18 months of this project. Dr. Venkataramani's participation had two main purposes: 1) to ensure smooth transfer of knowledge to Jungju Oh, the graduate student who is now working on this project, and 2) to ensure that there is no conflict and that there is synergy between the research on this project with the work Guru is doing on his own project. Dr. Venkataramani has stopped participating in our meetings because he was concerned that such participation and his subsequent co-authorship on publications may be perceived by his colleagues at GWU as a lack of research independence from his PhD advisor.

Dr. Sanjeev Kumar (previously at Intel, now at Facebook) has been participating in our weekly meetings during the first 12 months of the project. However, since joining Facebook his research interests and time commitments have changed, so he no longer participates in our regular meetings but does occasionally provide feedback on our work.

Dr. Sourav Roy (Freescale) has expressed interest in possible applications of our work in the embedded space. We are keeping Dr. Roy appraised of our progress, and will follow Dr. Roy's suggestion to also use embedded benchmarks in evaluating performance debugging techniques and tools that result from this work.

Activities and Findings

Research and Education Activities: (See PDF version submitted by PI at the end of the report)

Please see the attached file.

Findings: (See PDF version submitted by PI at the end of the report)

Please see the attached file.

Training and Development:

Jungju Oh and Sunjae Park, the two PhD students who worked on this project, have gained valuable skills for further research in computer architecture, including profficiency in using multiple simulation and program analysis tools (SESC simulator, Pin dynamic instrumentation, GCC compilation infrastructure) and deep understanding of concurrency and performance of parallel programs.

Outreach Activities:

The PI has participated as a judge in the Georgia Science and Engineering Fair, the state-level competition for elementary and high-school students interested in science and engineering. It is improtant for faculty at top research institutions to be involved in these types of competitions, not only to ensure that judging is done well, but also to provide students with an opportunity to interact with such faculty and be encouraged to pursue further education and careers in science and engineering disciplines.

Journal Publications

Venkataramani, Guru; Hughes, Christopher; Kumar, Sanjeev; Prvulovic, Milos, "DeFT: Design Space Exploration for On-the-Fly Detection of Coherence Misses", ACM Transactions on Architecture and Code Optimization (TACO), p. 8:1, vol. 8, (2011). Published, 10.1145/1970386.1970389

Books or Other One-time Publications

Ioannis Doudalis, Milos Prvulovic, "HARE: Hardware Assisted Reverse Execution", (2010). Conference Proceedings, Published

Collection: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)
Bibliography: ISBN 978-1-4244-5659-8

Ioannis Doudalis, Milos Prvulovic, "Euripus: A Flexible Unified Hardware Memory Checkpointing Accelerator for Bidirectional-Debugging and Reliability", (2012). Conference Proceedings, Published
Collection: Proceedings of the 39th IEEE/ACM International Symposium on Computer Architecture (ISCA)
Bibliography: ISBN 978-1-4503-1642-2
Also appears in SIGARCH Comput. Archit. News, DOI 10.1145/2366231.23371

Web/Internet Site

Other Specific Products

Contributions

Contributions within Discipline:

The contributions of this project to the area of Computer Architecture thus far are 1) a new programmer-centric definition of true and false sharing, together with a new understanding that different definitions might be appropriate for different goals (e.g. for design of new hardware or for design of software to run on existing hardware), 2) a novel method for pinpointing load imbalance problems in many-core execution, which can be used to identify code changes to allow programs to execute more efficiently on existing hardware, and also to guide design of new architectures that would alleviate the problems present in existing software, 3) a new method for reporting lock contention problems in many-core applications, along with specific suggestions on how to reduce this contention and expected performance improvement for each suggested change.

Contributions to Other Disciplines:

The work on load imbalance pinpointing has been published in the top conference in Software Engineering for two reasons 1) to facilitate broader use of the tools by presenting them to the software development community (who tend to monitor developments in software engineering conferences), 2) to create a 'bridge' between computer architecture and software engineering communities, which will need to work together on solving the increasingly-pressing problems of software development for many-core platforms.

Contributions to Human Resource Development:

Contributions to Resources for Research and Education:

The PI's understanding of parallel performance bottlenecks that resulted from this project has been used to revise projects in the CS 4290/6290 'High-Performance Computer Architecture', a co-taught senior undergraduate and introductory gradaute course in computer architecture. The revised projects are intended to enhance both 1) the students' understanding of the interaction between multi-threaded applicatons and multi- and many-core machines, and 2) improve the students' ability to produce parallel code that will run efficiently on current and future many-core platforms.

Contributions Beyond Science and Engineering:

Conference Proceedings

Oh, JJ;Hughes, CJ;Venkataramani, G;Prvulovic, M, LIME: A Framework for Debugging Load Imbalance in Multi-threaded Execution, "MAY 21-28, 2011", 2011 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), : 201-210 2011

Oh, J;Prvulovic, M;Zajic, A, TLSync: Support for Multiple Fast Barriers Using On-Chip Transmission Lines, "JUN 04-08, 2011", ISCA 2011: PROCEEDINGS OF THE 38TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, : 105-115 2011

Categories for which nothing is reported:

Any Web/Internet Site

Any Product

Contributions: To Any Human Resource Development

Contributions: To Any Beyond Science and Engineering

Final Report - Findings

for NSF Award 0903470, Performance Debugging Support for ManyCore Processors

PI: Milos Prvulovic, Georgia Institute of Technilogy

1 Introduction

In recent years, the number of cores available on a processor has increased rapidly, while the performance of an individual core has increased much more slowly. As a result, achieving a large performance improvement for applications now requires programmers to leverage the increased core count. This is often a very challenging problem, and many parallel applications end up suffering from *performance bugs* caused by *scalability limiters*. These prevent performance from improving as much as it should with more cores. Since we expect core counts to continue increasing for the foreseeable future, addressing scalability limiters is important for developing software that will obtain better performance on future hardware.

This project, jointly funded by SRC and NSF, investigated software and hardware mechanisms that automate significant parts of this performance/scalability debugging effort in order to give programmers accurate and actionable feedback about the scaling limiters present in their code. Scalability limiters are mostly caused by resource-related bottlenecks and by insufficient exposed parallelism in the application. The main resource-related bottlenecks are related to excessive cache misses, while insufficient parallelism is mostly manifested as threads waiting to complete a synchronization operation such as a lock (lock contention) or a barrier (load imbalance).

This report is organized as follows:

First, in Section 2, we present our comprehensive cache miss classification and reporting tool, which consists of three classification mechanisms - one that separates coherence and non-coherence misses, one that separates coherence misses into those caused by false and those caused by true sharing, and one that separates non-coherence misses into conflict and non-conflict (mostly capacity misses).

Second, in Sections 3 and 4, we present the two tools for analysis and reporting of synchronization-related limiters, i.e. limiters caused by waiting on synchronization operations. Among synchronization operations, barriers create scalability problems when there is *load imbalance*, i.e. when some threads finish their work faster than others and then must wait for those others at the barrier. Load imbalance analysis and reporting in an actionable way is accomplished by our tool called LIME (Section 3), which points to specific code points (lines of code) where load imbalance is introduced, and reports what type of load imbalance is introduced at each code point. Lock contention is one of the more common scaling limiter because locks are nearly ubiquitous in all multi-threaded software where they can create excessive *lock contention*, where threads must wait at the entry point into a critical section while other threads make their way through that critical section one at a time. Our approach to lock contention analysis and reporting is presented in Section 4. Another type of synchronization are conditionals, but they tend to be used less often and usually in application-specific patterns, so we have not (yet) created specific tools and mechanisms for analysis and reporting of conditionals-related synchronization waiting.

Section 5 presents our efforts to use the research results of this project in class projects, to facilitate student's understanding of scaling problems and many-core performance in general.

Finally, Section 6 presents our approach to integrated analysis and reporting of scaling limiters that combines these three tools, along with our overall conclusions and potential directions future work.

2 Cache Miss Classification and Reporting

Modern processors have several levels of caches to facilitate speedy access to data that are accessed multiple times within short time intervals (temporal locality) as well as data that are accessed within relatively close regions (spatial locality). Memory accesses that do not have data in caches (also known as cache misses) lead to poor application performance and it is necessary to understand their behaviors to minimize their effects.

The infrastructure most commonly used by application writers to diagnose performance issues, for both serial and parallel applications, are performance counters. These are hardware counters that enable collecting statistics from a production environment using real inputs. Today's processors support counting a large number of events with performance counters, making them a powerful and general tool [26]. Hardware support for more sophisticated performance counters [45], attribution of events to instructions [13], and on-the-fly processing of profiling data [38, 39, 54] are active research topics. However, performance counters and other profiling infrastructure currently only monitor hardware events that are relatively simple to detect and classify (e.g., branch mispredictions, retired instructions, cache misses, etc.). While this basic information is sufficient to diagnose some performance limiters such as the saturation of external memory bandwidth, it is not sufficient to diagnose limiters resulting from more complex behavior (e.g., it does not point to a particular type of cache miss in a specific region of code as a performance limiter).

This section describes CacheDoc, a comprehensive Cache Miss Classifier and explore several designs that classify cache misses into specific categories. We first provide some background on the cache miss classification problem (Section 2.1). Section 2.2 then provides a new, programmer-friendly, definition of false sharing, along with real-world examples that help understand the concept of false sharing and

the need for a new definition. Section 2.3 starts with ideal/off-line coherence miss classification and explores several practical designs that offer a spectrum of cost-accuracy tradeoffs. Section 2.4 provides a similar ideal-to-practical analysis for classification of replacement misses. Section 2.5 combpresents results that combine practical classification of coherence and replacement misses into a comprehensive cache miss classification scheme called CacheDoc. Finally, Section 2.6 discusses related work, and Section 2.7 summarizes our findings.

2.1 Background

With the growing popularity of modern day multi-core architectures, tapping their potential requires efficient use of the multiple cores on the chip. Applications must be written so that their performances scale linearly, or as close to linearly as possible, with the number of available cores. This is often extremely challenging, even when the underlying application is amenable to parallelization. Parallel applications have a number of possible performance problems that either do not exist in serial applications or that can be aggravated by the sharing of resources.

Processor vendors understand that making processors with increased raw performance is not sufficient: customers must see a real performance difference. This requires that software vendors write their applications to take advantage of the increased raw performance. This is not a trivial task, and therefore, most processor vendors currently support a number of performance counters to aid this task [26]. Performance counters aid programmers in finding and eliminating performance bottlenecks in their code (i.e., performance debugging). Since it is much harder to realize the performance potential of multi-core and many-core processors, processor vendors should be willing to invest some additional resources to aid in performance debugging issues introduced with multi-core and many-core processors. Ideally, these resources would include performance counters for additional multi-core and many-core events, and hardware to detect those events.

A natural question to ask is whether hardware support brings significant value to performance debugging given the wide variety of software-only performance debugging tools available. For at least three aspects, the answer is "absolutely." Tools that interface with the performance counters are extremely popular because they 1) can collect information on an enormous variety of performance bottlenecks and present it in a usable way (i.e., pinpoint problems in the source code), 2) are very fast, and 3) are very accurate.

In contrast, software-only tools 1) typically only measure a small set of events since they must trade off execution time overhead and the amount of information collected, 2) are generally slow even when only measuring a small number of events (e.g., SM-prof [7] reports up to 3000$\times$ slowdown depending on the amount of shared data references), and 3) are not reliably accurate because in many cases they cannot model hardware perfectly and/or perturb the program execution by interleaving software instrumentation into that execution. Having a tool that is fast, accurate, and can measure a number of events of interest is extremely valuable.

Memory accesses that miss in caches frequently consume additional latency and degrade program performance. While cache misses certainly affect the performance in single core machines, their effects are even more pronounced in multi-core processors. A variety of memory system behaviors that result in poor application scaling manifest themselves as additional cache misses. Therefore, determining the source of the additional misses can be of great help to a programmer. The "source" of a cache miss includes both the underlying reason why the hardware did not have the line in the cache as well as the place in the code that triggered the miss.

Hardware may incur additional cache misses for parallelized applications for a number of reasons. The most common ones for shared memory systems are: 1) Reduced temporal and spatial locality from input data partitioning, 2) inter-thread communication, via both true sharing (i.e., intentional communication) and false sharing (i.e., unintentional communication), and 3) destructive sharing (i.e., additional conflict misses from cache sharing).

False sharing and destructive sharing, in particular, are issues that often surprise programmers; without sufficient knowledge of the underlying hardware, the presence of these misses is mysterious and even frustrating. They can also have a devastating impact on parallel scalability (Section 2.2). For example, programmers often use an array of counters or accumulators (one per-thread) when parallelizing reductions. Without appropriate padding, if these structures are frequently accessed, the application will incur a huge increase in cache misses due to false sharing. Another frequent programmer behavior is partitioning a data structure into power-of-two-sized chunks. If the chunks are aligned in a shared cache (i.e., map to the same sets), and the cache's associativity is not sufficient, the application will incur a huge increase in cache misses due to destructive sharing [21]. This also frequently occurs if threads' stacks are power-of-two aligned.

One way to help distinguish between the above three causes of cache misses is to classify misses using the common categories: 1) compulsory, 2) capacity, 3) conflict, and 4) coherence [22]. A cold miss occurs when the requested block has never been in the cache before. Capacity and conflict misses are collectively called replacement misses. These misses occur when a block is re-accessed after it has already been fetched into the cache and then evicted to accommodate another block. Capacity misses are replacement misses due to limited cache size. Conflict misses are caused by limited cache associativity; these would be hits in a fully associative cache of the same size. Finally, coherence misses are caused by sharing of data between cores. These misses occur when a block is invalidated or downgraded in a core's cache, and then re-accessed by that core. Unlike the other three classes of misses (compulsory, capacity, and conflict), a coherence miss finds the block in the cache, but the block is in a non-usable state: a read finds the block in an invalid state, or a write finds the block in invalid or shared states.

Classifying cache misses into these categories can help identify the root cause of the misses. For example, a large jump in conflict misses in a shared cache between a single-threaded run and a multi-threaded run likely indicates destructive sharing. It is also helpful to additionally classify coherence misses as being caused by true or false sharing since these have fundamentally different sources, and are addressed by the programmer in different ways. Cache miss classification helps programmers to use techniques, such as array grouping to reduce communication delays [46] or cache conscious data placement [9, 10], to minimize the effect of such misses. Although our aim is to primarily help programmers, we derive other potential benefits from identifying various types of cache misses. For example, victim caches can choose to store blocks that frequently suffer conflict misses [11]. Prefetchers can improve performance by bringing in blocks that suffer a large number of capacity misses.

We note that performance of applications can be improved only if programmers, compilers, and/or dynamic optimizers carefully apply appropriate fixes to reduce the number of cache misses in the program. Different types of cache misses require different types of fixes, and knowing the dominant type of cache misses is helpful when deciding on an approach to fixing them. For example, conflict misses and false sharing misses are frequently addressed through padding the memory, while true sharing misses might require fundamental redesign of the algorithm. Although there is no doubt that the effort required differs depending on the type of miss, not all misses of a particular type are amenable to the same fixes. For example, fixing false sharing misses by padding a data array may introduce additional cache misses due to conflict with other cache blocks. Hence, depending on the type of miss and its surrounding access patterns, potential performance improvement is likely to vary across different cases. Nevertheless, identifying the sources and causes (type) of these cache misses helps programmers to understand program behavior and to decide which techniques to use for improving program performance.

There have been several proposals for cache miss classification off-line or in architectural simulators [22, 37, 50]. A pure software-based scheme to dynamically classify cache misses can degrade the application performance by several orders of magnitude [7]. This is unacceptable not only because it is slow but also because it can significantly alter the execution path of a parallel program, minimizing the utility of the generated cache miss classification. Collins et al. [11] have proposed a hardware scheme that identifies conflict misses in a cost-effective manner. However, their definition of a conflict miss differs from the classical definition and is primarily directed towards helping prefetchers and victim caches. To our knowledge, our scheme (which we named CacheDoc) is the first to provide a comprehensive cache miss classification mechanism that can be performed on-the-fly to drive performance counters and hardware-assisted profiling.

2.2 False Sharing and its Implications on Scalability

In Section 2.1, we described the need to classify cache misses into compulsory, replacement and coherence misses. Coherence misses, in particular, are becoming increasingly relevant for multi-core processors and needs to be better understood by programmers who are much more familiar with compulsory and replacement misses that occur in single core processors. In this section, we provide a programmer-centric definition of false sharing and true sharing misses and then discuss several real-world examples where false sharing misses occur and how removing these misses improves scalability.

2.2.1. Definition. In cache coherent CMPs, data sharing between threads primarily manifests itself as coherence misses which can further be classified as true sharing and false sharing misses.

True sharing misses are a consequence of actual data sharing among cores and are intuitive to most programmers, e.g., a consumer (reader) of the data must suffer a cache miss to obtain the updated version of the data from the producer (writer). The scalability issues stemming from true sharing misses can typically be addressed only by changing the algorithm, distribution of the work among cores, or synchronization and timing.

In contrast, false sharing misses are an artifact of data placement and a cache block holding multiple data items. Scalability issues arising from false sharing are often relatively easy to alleviate by changing alignment or adding padding between affected data items. A false sharing miss occurs if a block contains two data items (X and Y), core A accesses item X, core B uses item Y and invalidates the block from A's cache, and then core A accesses item X again. The resulting cache miss is a false sharing miss because that access would be a cache hit if X and Y were in different cache blocks. Note that, if after the coherence miss on an item X, core A accesses item Y before the block is evicted or invalidated again, the miss is in fact a true sharing miss. As shown in Figure 1, in this situation, the cache miss is not avoided by placing X and Y in separate blocks, and hence, the miss on X is not a false sharing miss. This definition of false sharing is based on Dubois et al. [15], and it differs from earlier definitions [17, 50] in that it attempts to classify coherence misses according to whether they are "essential" (true sharing) or "non-essential" (false sharing). We adopt this definition as a baseline because it more accurately captures whether or not the miss can be eliminated (not just traded for another miss) by separating data items into different blocks which, in the end, is what really matters for the programmer's performance debugging efforts.

2.2.2. Real-World examples. We provide examples, taken from code written by experienced programmers, to illustrate some common situations where false sharing occurs and its sometimes devastating impact on parallel scalability.

Our first example involves an array of private counters or accumulators (one per-thread), which is often used when parallelizing reductions as shown in Figure 2. There is no true sharing in this code because each thread is reading and writing a unique array element. False sharing occurs when two threads' counters lie in the same cache block. This kind of code is common in web search, fluid simulation, and human body tracking applications [14]. A common fix is to add padding around each counter. We provide a real-world example from a loop in the *One_Newton_Step_Toward_Steady_State_CG_Helper_II()* function from the facesim benchmark in the PARSEC-1.0 suite. The benchmark authors spent multiple days identifying false sharing from this loop as the primary source of performance problems [14]. We ran the facesim benchmark with the native input on an 8-core Intel Xeon machine and observed that without padding, false sharing limits the benchmark's parallel scaling to 4× (Section 2.3.2). After adding padding, the benchmark achieves linear scaling (8×). A profiling tool that automatically identifies and reports false sharing misses would have greatly helped the programmer.

Our second example involves an indirectly accessed data array, as shown in Figure 3 and often occurs in histogram computation, used in image processing applications and in some implementations of radix sort [14]. The pattern of indirections is input-dependent, so the programmer and compiler cannot predetermine how many accesses will occur to each element, and which threads will perform them. This example involves both true and false sharing. True sharing occurs when two threads update the same element. False sharing occurs when two threads access two different elements in the same cache block, which occurs much more frequently. For example, with 64-byte blocks and 4-byte elements, a block contains 16 elements; with a completely random access pattern, false sharing would occur 15 times more likely than true sharing. A common fix is to either add padding around each element or use privatization (use a separate array for each thread and then

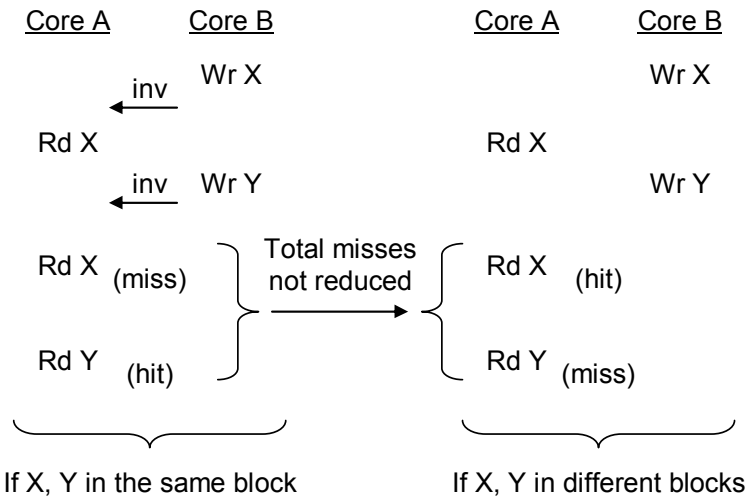


Figure 1. The miss on X is a not a false sharing miss — it is only replaced by another miss if X and Y are placed in separate blocks.

```
// Each thread processes its own
// sequence of input elements
int partial_result[NUM_THREADS];
// This is the work done in parallel
...
int input_element=...;
partial_result[thread_id] += input_element;
...
```

Figure 2. A parallel reduction showing false sharing on an array of counters (one-per-thread). The merging of partial results is omitted for space.

merge partial results at the end). Without privatization, a histogram benchmark from a real-world image processing application achieves only a 2× parallel speedup when run on a 16-core Intel Xeon [26] machine. With privatization, the benchmark achieves near-linear scalability.

Our final example of false sharing involves finely partitioned arrays, such as one might find in red-black Gauss-Seidel computations as shown in Figure 4. In many applications, a data array is partitioned such that each partition will only be written to by a single thread. However, when updating elements around the boundary of a partition, a thread sometimes needs to read data across the boundary (i.e., from another partition). In our example, synchronization on each element is avoided by treating the data as a checkerboard, with alternating red and black elements. Even-numbered passes update red cells, and odd-numbered passes update black cells. An update involves reading the Manhattan-adjacent neighbors, which are guaranteed to be a different color than the cell being updated. Thus, even if those neighbors belong to another partition, they will not be updated during the current pass.

```
// Count occurrences of values in parallel
// Each thread processes its own range of
// input elements, updating the shared
// occurrence count for each element's value

#define MAXIMUM 255
int counter_array[MAXIMUM+1];
// This is the work done in parallel
...
int input_element=...;
  counter_array[input_element]++;
...
```

Figure 3. A parallel histogram computation illustrating false sharing on an indirectly accessed array. Locking of counter_array elements is omitted for brevity.

```
// The task of each thread is to update one row of the grid
...
for (i = (iteration%2);i < width;i += 2) {
  float val = grid[task_id  ][i  ] + grid[task_id  ][i-1] +
              grid[task_id  ][i+1] + grid[task_id-1][i  ] +
              grid[task_id+1][i  ];
  grid[task_id][i] = val / 5.0;
}
...
```

Figure 4. A red-black Gauss-Seidel-style array update showing false sharing on a finely partitioned array.

Both true and false sharing occurs in this example. The first time a thread accesses a cache block across a partition boundary, it is reading data written by another thread during the previous pass. Thus, it incurs a true sharing miss. However, if that other thread is actively updating elements in the same cache block, this will trigger additional misses that are all due to false sharing. This situation is most likely when partitions are small; for example, if a parallel task is to update one row, and the tasks are distributed to threads round-robin. This kind of computation is common in scientific codes (i.e., applications that involve solving systems of differential equations) [14]. We ran an early real-world implementation[1] of red-black Gauss-Seidel with the above task distribution on an 8-core Intel Xeon processor. Parallel scaling is limited to 3x. Padding around each cell can eliminate false sharing, but with significant loss in spacial locality. A better solution is to group multiple rows together to be processed by the same thread, or to use a two separate arrays, one for red and one for black cells. In our case, using separate arrays and grouping rows improved the scaling to almost 6x.

2.3 Classification of Coherence Misses

In addition to compulsory, capacity, and conflict misses that occur in both single-core and multi-core processors, multi-cores also suffer from coherence misses, which occur as a result of coherence actions between private caches of different cores.

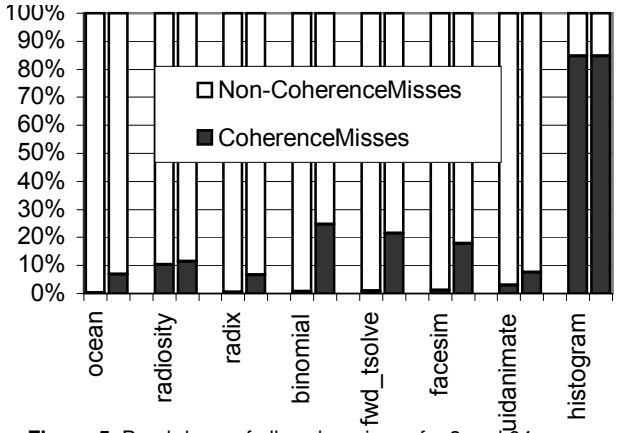


Figure 5. Breakdown of all cache misses for 8 and 64 cores.

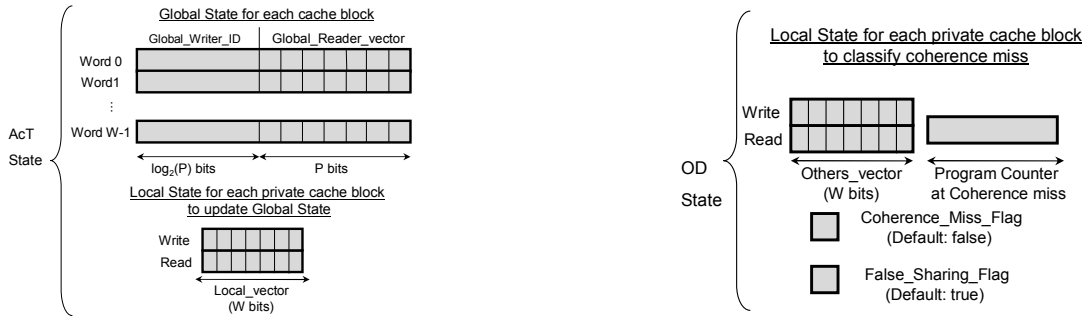
Figure 5 shows the breakdown of cache misses into coherence and non-coherence misses as we increase the number of cores from 8 to 64 for benchmarks with the most data sharing. With more cores, coherence misses represent an increasing percentage of all cache misses. As a result, performance issues related to true and false sharing become worse when more cores are used, and often become a scaling limiter. It is important to provide developers with tools and mechanisms that detect the presence of these misses, distinguish between the two types, and identify their sources.

In this section, we first describe the relatively simple mechanism for distinguishing coherence misses from non-coherence misses, then discuss an Ideal False Sharing Detector (I-FSD) mechanism that further classifies coherence misses into those caused by false sharing and those caused by true sharing. We then contrast I-FSD with the algorithm proposed by Dubois et al. [15] and show the key differences between the two schemes. We show why I-FSD is impractical to implement in real hardware and explore several practical hardware designs for coherence miss classification.

2.3.1. Identification of Coherence Misses. Coherence misses can be distinguished from other (cold, coherence, or conflict) misses by checking if the cache block is already present in the cache. For non-coherence misses, the block either never was in the cache (cold miss) or was replaced by another block (capacity or conflict miss). In contrast, a coherence miss occurs when the block was *invalidated* or *downgraded* to allow another core to cache and access that block. Coherence misses are easily detected with a minor modification to the existing cache

[1]We gratefully acknowledge researchers at Applications Research Lab, Intel Corporation for providing us with an early version of Gauss-Seidel benchmark.

```
Input: P=total number of cores, W=number of words in a cache block
       A=current data address, B=cache block holding address A
       c=current core
```




```
ALGORITHM ACCESS_TRACKING
  IF(Coherence Miss to B)
    WAIT ON current sharers to update Global_Writer_ID and Global_Reader_vector
  IF(Write access to A)
    Local_vector_write[A]=1;
    Local_vector_read[A]=0;
  IF(Read access to A)
    Local_vector_read[A]=1;
  IF(Invalidation/Downgrade/Replacement request for B)
    FOR(each word address K in B) DO
      IF(Local_vector_write[K]==1)
        Global_Writer_ID=c;
        Global_Reader_vector={0};
      IF(Local_vector_read[K]==1)
        Global_Reader_vector[c]=1;
    DONE

ALGORITHM OVERLAP_DETECTION
  IF(Coherence Miss to B)
    Coherence_Miss_flag=true;
    False_Sharing_flag=true; //Reset on seeing first true sharing on the block
    Miss_PC=PC;
    Others_vector_write=Others_vector_read={0};
    FOR(each word address K in B) DO
      IF(Global_Writer_ID!=c AND Global_Reader_vector[c]!=1)
        Others_vector_write[K]=1;
      IF(a bit other than c is set in Global_Reader_vector)
        Others_vector_read[K]=1;
    DONE
  IF(Write access to A)
    IF(Others_vector_write[A]==1 OR Others_vector_read[A]==1)
      False_Sharing_Flag=false;
  IF(Read access to A)
    IF(Others_vector_write[A]==1)
      False_Sharing_Flag=false;
  IF(Invalidation/Downgrade/Replacement request for B)
    IF(Coherence_Miss_flag==true)
      //Output Miss_PC and False_Sharing_flag to Profiler
```

Figure 6. I-FSD Algorithm

lookup procedure. A cache miss is detected as a coherence miss if a block does have a matching tag but does not have a valid state. Conversely, if no block with a matching tag is found, we have a non-coherence miss.

2.3.2. Ideal False Sharing Detector (I-FSD). We use the *programmer-centric* definition of false sharing described in Section 2.2. Basically, a coherence miss is a false sharing miss if none of the memory accesses (starting with the memory access causing the miss up until the block is invalidated or downgraded or replaced) happen on data that were accessed by other cores. Coherence misses are classified into false sharing misses and true sharing misses by our I-FSD algorithm [51] shown in Figure 6.

Coherence miss classification involves two distinct phases namely:

- *Access Tracking (AcT)*: From the time, a cache block is invalidated or downgraded in a core's cache, until the time when coherence miss happens, we need information about which words were read from or written to by other cores.
- *Overlap Detection (OD)*: From the time of coherence miss until when the block is invalidated/downgraded/replaced, we need to determine whether an access to any word in the block overlaps with an access by another core. True sharing access happens when

Table 1. Program Counters and the corresponding number of false sharing misses reported in Facesim Benchmark.

Program Counter	False Sharing Count
4cc084	9921186
4cc074	7803903
53eb34	251457
53f2ac	214323
53f5d8	201885
...	
Total=	18854631

a memory word is either i) written and read by two different cores or ii) written by two different cores. If we do not detect any true sharing access, then the coherence miss is a false sharing miss. Note that read by two different cores do not cause invalidation or downgrade, hence coherence misses don't occur in these cases.

The two phases, namely AcT and OD, are *temporally disjoint* phases for coherence miss classification. AcT phase occurs between the time a cache block is invalidated or downgraded from a cache and when the coherence miss happens (that is, when the cache block is accessed again by the core). OD phase occurs between the time of coherence miss and when the cache block is invalidated or downgraded or replaced.

In order to track these phases accurately, I-FSD maintains the following data structures namely:

- `AcT state` which has the following information: i) per-word[2] Global State that tracks last writer core and subsequent readers until the next write. ii) Two bit vectors for every private cache block that track read and write accesses by the local core during *OD* to update the Global State at the time of cache block invalidation/downgrade/replacement. This is needed for other cores to perform their *AcT* correctly.

- `OD state` which has the following information: i) Two bit vectors for every private cache block which records reads and writes performed by other cores to the cache block prior to the time of coherence miss. ii) Program Counter at the time of coherence miss for attribution to the program code that caused the miss. iii) Two flags, one to track whether the cache block suffered coherence miss and the other to track whether the all the accesses to the cache block are false sharing accesses.

`PROC ACCESS_TRACKING` implements the AcT phase. Information regarding every read and write access by a core is accumulated in Local_Vectors and at the time of cache block eviction, the information is transmitted to update the Global State.

`PROC OVERLAP_DETECTION` implements the OD phase. At the time of coherence miss, the information from Global State is captured into Others_vectors. On every read and write, this information is used to determine if the current memory access results in true sharing. A read access is a true sharing access if the word's last write was not done by this core and if this core has not already read the word since it was last written. A write access is a true sharing access if the word's last write was not done by this core or if it was read by another core since it was last written. A coherence miss is classified as a true sharing miss if any access, starting with the one that causes the miss and ending with the subsequent replacement or invalidation of the block, is a true-sharing access. Conversely, the miss is categorized as a false sharing miss if no true sharing access occurs in this interval.

We present an example case study for Facesim benchmark from Parsec-1.0 suite [5] to show how our I-FSD algorithm correctly captures false sharing misses. Since the existing version has already been tested and fine-tuned for performance before release, we introduce false sharing explicitly. This is done to backtrack the original efforts by programmers to identify false sharing except that they did not have access to our tool during development. We present our results to examine the effectiveness of our tool in identifying the regions of code where false sharing misses occur.

Inside the *One_Newton_Step_Toward_Steady_State_CG_Helper_II()* function, we identify a tight loop where there are two accumulator variables *rho_new* and *supernorm* whose values are computed and updated in every iteration using the input elements. Since, each thread privately owns the accumulators and are stored in a shared array, all coherence misses that result from accessing them are false sharing misses. This limits scalability to 4x on a 8-core machine.

When we run Facesim through our I-FSD tool, we obtain the results shown in Table 1. Clearly, the static instructions at addresses `4cc084` and `4cc074` are responsible for 94% of false sharing coherence misses in the program and approximately 24.5% of all cache misses. Using the *addr2line* utility from GCC toolchain [18], we identify the part of the program where these addresses are located and find that they point to lines inside the loop where the accumulators *rho_new* and *supernorm* are being updated (Figure 7). We change the loop to accumulate the results locally into *local_new_rho* and *local_supernorm* and update shared variables *new_rho* and *supernorm* after the loop is complete. This avoids the false sharing coherence misses on these accumulators and results in near-linear (near-8x) speedup on an 8-core machine.

[2]Granularity is a matter of cost-performance trade-off. We did not observe significant coherence activity on sub-word accesses. Hence we chose word-granularity for our work. We note that byte-granularity can be achieved with the same algorithm with relatively straightforward modifications.

```
void One_Newton_Step_Toward_Steady_State_CG_Helper_II(...)
{
  ...
  for(int i=1;i<=dX.m;i++)
  {
   dX(i)+=alpha*S(i);
   R(i)+=alpha*negative_Q(i);
   double s2=R(i).Magnitude_Squared();
   rho_new+=s2;
   supnorm=max(supnorm,s2);
  }

  //False Sharing Fix
  // Locally accumulate the counters and update later
  //for(int i=1;i<=dX.m;i++)
  //{
  // dX(i)+=alpha*S(i);
  // R(i)+=alpha*negative_Q(i);
  // double s2=R(i).Magnitude_Squared();
  // local_rho_new+=s2;
  // local_supnorm=max(local_supnorm,s2);
  //}
  //rho_new=local_rho_new;
  //supnorm=local_supnorm;

  ...
}
```

Figure 7. False Sharing Misses in Facesim benchmark.

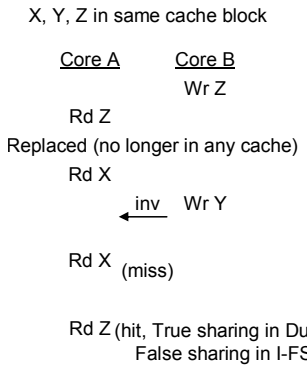


Figure 8. Accurate classification requires us to keep access information even for blocks that are no longer in any cache.

2.3.3. Comparison with Dubois' scheme. To our knowledge, Dubois et al. [15] was the first proposal to classify coherence misses as essential (true sharing) and non-essential (false sharing). They propose separate algorithms for different types of cache coherence protocols. We use the invalidate-based protocol (MESI) implementation described in [15] for our comparison. This implementation maintains a *stale bit* for every cache word. When a write happens on a word, the caches that currently have the word clear the stale bit to indicate that a new value is produced. A subsequent read of this word denotes consumption of "fresh value" produced by a write and is treated as true sharing. Also, the stale bit is set to denote that any further read (without an intervening write) will only consume a stale value.

There are two key differences between I-FSD and Dubois scheme. First, writers (producers) never classify coherence misses and all write misses are assumed to be non-essential misses. Hence, a portion of true sharing misses that can classified on the producer's side are classified as non-essential misses in Dubois' scheme. Whereas, I-FSD classifies coherence misses both on the producer and consumer sides. In a *single producer-multiple consumer* pattern (See Gauss-Siedel example in Section 2.2.2) and *multiple producer-multiple consumer* pattern (See histogram example in Section 2.2.2), coherence misses are highly likely to happen in multiple cores (producer and the respective consumers). Dubois' scheme could underestimate the effect of false sharing in such situations and the programmer might consequently ignore or oversee the lines of code involved in such patterns because of lack of understanding of its impact.

Second, on every cache block replacement, the stale bits are cleared. This is because, the *staleness* of values tracked by the consumers are no longer relevant by virtue of being replaced by another cache block. This results in loss of history about the "staleness of values" (information that a consumer had already read the value from the cache word). When the cache word is subsequently read by the core, the stale bit is zero and hence, it is treated as true sharing by Dubois' scheme. Figure 8 shows an example where the cache block having items X, Y and Z has been replaced by both cores A and B and when core A suffers a coherence miss, Dubois' scheme classifies "Read Z" as true

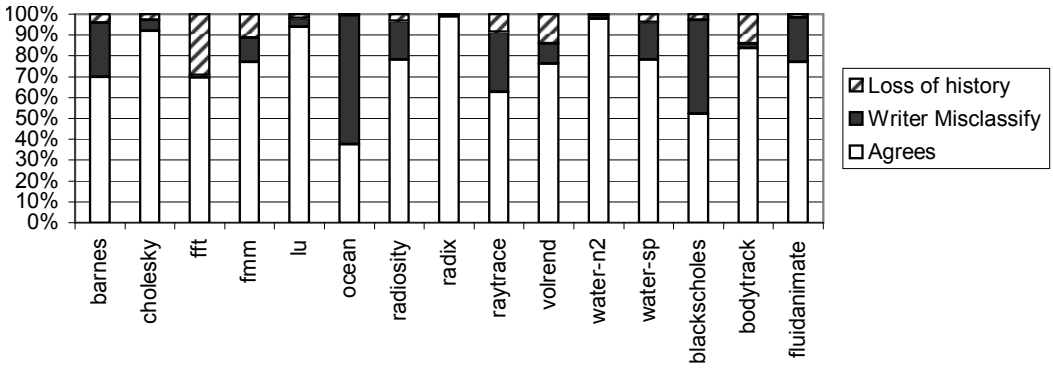


Figure 9. Comparison between I-FSD and Dubois' scheme. Each bar has three portions- bottom portion shows the percentage of times there is agreement between two schemes, middle portion shows percentage times Dubois' scheme misclassifies as false sharing (non-essential) on the producer side and the top portion shows the percentage times Dubois' scheme misclassifies as true sharing because of loss of history regarding *staleness* of values.

sharing access. This is because, the information regarding the "staleness" of value Z has been lost during cache block replacement. However I-FSD remembers that the value of Z had already been read by core A and would correctly classify the access as false sharing.

Figure 9 shows the results of our experiments. Each benchmark has three portions. The bottom portion of each bar shows the percentage of coherence miss classifications that both Dubois' and I-FSD schemes agree. In many benchmarks, the two schemes agree at least 75% of the time except fft, ocean, raytrace and blackscholes. The middle portion of each bar shows the percentage of times the writer classifies coherence misses as non-essential (false sharing) in Dubois' scheme whereas I-FSD detects true sharing on the writer side. A large portion of such misclassifications occur in benchmarks such as ocean (62%), blackscholes (45%), raytrace (28%) and fluidanimate (21%). The top portion of each bar shows the percentage of times Dubois' scheme detects essential (true sharing) whereas I-FSD detects false sharing. Such cases occur when *stale bits* are cleared on cache block replacement. This results in loss of history information regarding "staleness" of values that are already being read. Such misclassifications are frequent in a few benchmarks such as fft(29%), volrend(14%) and bodytrack (14%). I-FSD correctly detects false sharing because it maintains the access history information in global state (similar to a central repository).

2.3.4. Suitability of I-FSD for On-Line Miss Classification. There are a number of problems that make the Ideal False Sharing Detector unsuitable for on-line implementation needed to drive hardware performance counters or other hardware performance debugging and attribution mechanisms. The most significant of these problems are:

1. The I-FSD has a high implementation cost which does not scale well. The *per-word* Global State for the I-FSD structure shown in Figure 6 is $P + log_2 P$ where P is the number of cores. With 4 cores and 32-bit words this state represents a 19% storage overhead, and with 32 cores the storage overhead is already 116%.

2. The state needed for the I-FSD can be very large (requires keeping state for all words ever touched). It may be tempting to not keep track state for a word that is no longer present in any core's cache. However, information about currently evicted blocks might be relevant for classifying future coherence misses to these blocks (See example in Figure 8).

3. Changes are needed to the underlying cache coherence protocol and extra network traffic is required to update the I-FSD state and/or propagate it to the cores that need it to classify their coherence misses. In particular, Global State information for a memory word should be kept in a central repository (could also be stored in the last level shared cache or memory). As seen in Figure 6, the coherence protocol needs to be modified to trigger reads and writes of the global state information at appropriate times. To enforce that, the central repository needs to ensure updates sent by the cores replacing/invalidating/downgrading the cache block are reflected in the Global State before the core suffering the coherence miss reads the information from the central repository and updates its local state to perform *OD*.

4. A coherence miss may be classified as a true or false sharing miss many cycles after the instruction causing the miss has retired from the core's pipeline. This delay in classification makes it more difficult and costly to attribute false and true sharing misses to particular instructions, especially if performance counters support precise exceptions for events, such as PEBS (Precise Event Based Sampling) mechanism available in recent Intel processors [2]. For accurate attribution, the PC of the instruction that caused a coherence miss must be kept until the miss is eventually classified. This increases the cost of the classification mechanism, and also makes it more difficult for profilers to extract other information about the miss (e.g., what was the data address or value) [2].

To provide a realistically implementable scheme for on-line classification of coherence misses, we need to overcome some (preferably all) of the above problems, while sacrificing as little classification accuracy as possible. When choosing which aspects of classification accuracy to sacrifice, we keep in mind the primary purpose of our classification mechanism: giving the programmer an idea of how much performance is affected by true and false sharing cache misses, and pinpointing the instructions (and from there, lines of code) that suffer most of these

misses. Armed with this, the programmer should be able to make an informed decision about how best to reduce the performance impact of these misses.

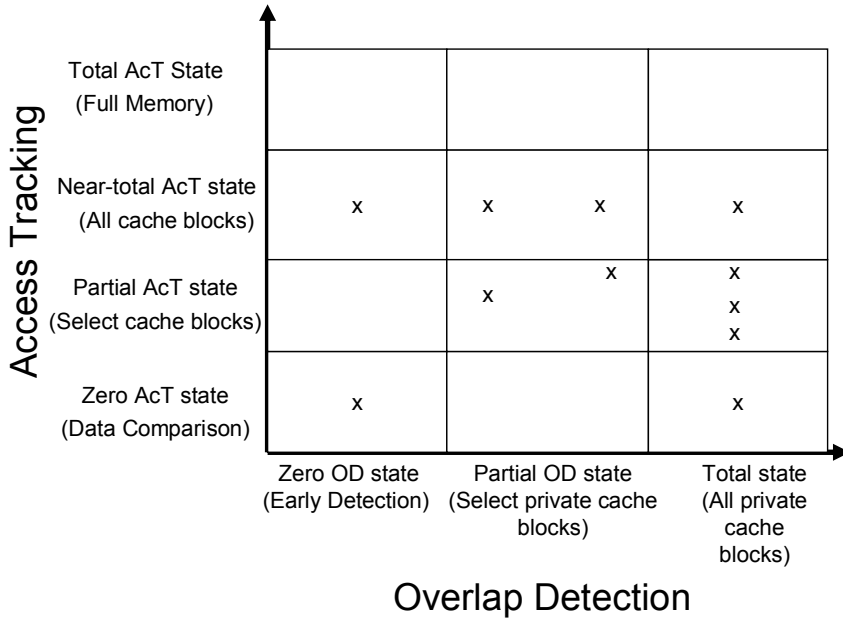


Figure 10. Design Space for coherence miss classification mechanisms. Gradations of AcT state is on the vertical axis and OD state is on the horizontal axis. Hardware costs increase as we move up or towards the right. Design points marked 'x' are examined in our evaluations.

2.3.5. Practical False Sharing Detectors. We propose and explore a range of design choices that trade-off accuracy for lower hardware costs. Section 2.3.4 enumerated a number of issues that limit I-FSD from being implemented in real machines. The key to overcoming these problems is to reduce the amounts of both global and local state. To achieve this, we make use of the fact that there are two distinct phases in Coherence miss classification namely *AcT* and *OD* (Section 2.3.2). The data structures needed by these two phases are independent and hence the mechanisms needed for the two phases can be designed independently. We call these mechanisms as practical FSD.

As shown in Figure 6, the Global State along with local access vectors (that track reads and writes by the local core to eventually update the global state) are needed to implement the *Access Tracking* phase correctly. We call this `AcT state`. Rest of the local state (namely Read and Write vectors of others, flags and the program counter at coherence miss) are used to implement the *Overlap Detection* phase. We call this `OD state`. Figure 10 shows the design space of coherence miss classification mechanisms defined by how much state is kept for each of these two phases.

2.3.6. Design Choices for Access Tracking. The most expensive AcT scheme (top row) maintains the AcT state for the entire memory throughout the program execution. We call this *Total AcT State*. As described in Section 2.3.4, the amount of global state grows proportional to the number of cores and the memory overhead increases at a super-linear rate with the number of cores. Hence this solution is impractical in real hardware.

The next most aggressive AcT implementation (second from top) maintains AcT state only for cache blocks in the on-die caches. We call this *Near-Total AcT state*. This scheme can only lead to misclassification of coherence misses to blocks that are evicted from all of the on-die caches between an invalidation/downgrade and the subsequent coherence miss. Evictions of such lines are highly likely to be a small fraction of the evictions, and thus this event should be quite uncommon. The state can be kept with the lowest level shared cache, for systems that have one. For chips with only private caches, and a separate mechanism for maintaining coherence such as a directory, the AcT state can be kept along with the coherence state for the core.

The third most aggressive AcT implementation (third from top) maintains Global State only for a limited number of cache blocks in the on-die caches. We call this *Partial AcT state*. This scheme can lead to misclassification of coherence misses if the cache block currently suffering coherence miss is not tracked globally. In order to select the subset of cache blocks for tracking global state, as a first order approximation, we could choose only blocks that currently occupy the last level private caches that are kept coherent. Furthermore, not all blocks in coherent caches are involved in sharing. Hence, we have opportunities to further restrict the number of entries to be tracked. To implement Partial State, we maintain a global pool where entries are allocated on demand to track select cache blocks. An entry is created when a cache block in any of the coherent caches is either invalidated or downgraded as they are likely to suffer from coherence misses. Subsequent accesses by various cores to the block are tracked and updated as described in I-FSD algorithm. Since the global pool holds a limited number of entries, it needs a policy to replace existing entries to make room for incoming blocks. We pick a modified version of pseudo-LRU replacement policy called NkMRU(Not 'k' Most Recently Used) replacement policy. Basically, it chooses a random entry for replacement that is not among the

'k' most-recently-used cache blocks. It should be noted that not all invalidated or downgraded cache blocks will eventually suffer coherence misses. Hence promoting every incoming block into the k most-recently-used set could result in losing valuable information about blocks that frequently suffer from coherence misses. To prevent this effect, we promote a cache block entry into the k most-recently-used set only when the cache block actually suffers a coherence miss in one of the caches.

The final and least aggressive AcT implementation (bottom row) does not maintain any AcT state and *infers* false and true sharing locally by comparing the stale value of the cache block with the incoming values. This technique of detecting false sharing through data comparison was used by Coherence Decoupling [23] to save cache access latency due to false sharing misses. If the data value has changed, then true sharing is detected, otherwise, the coherence miss is attributed to false sharing. We call this *Zero AcT state*. This scheme could misclassify in two situations: i) Silent stores [33] do not change the data value and hence, it is impossible to detect true sharing in such situations. However, classifying silent stores itself is a murky area. Whether they contribute to true or false sharing depends on specific situations. For example, lock variables have an initial value of zero. A core grabs a lock by setting it to one. Later when the lock is freed, it deposits a value of zero back. An external core does not see change in lock value and effectively sees a value of zero before and after the core operated on the lock. Even though lock sharing is a form of true sharing, detecting false sharing through data comparison would miss this effect. At the same time, other silent writes that simply do not communicate any new value can be eliminated through algorithmic changes, ii) While readers would be able to detect change in data value due to an external write, writers do not detect true sharing as readers do not modify data. As long as coherence misses are classified on the consumer side correctly, the programmer would still get an accurate picture of false sharing effects in the program. Note that, there is no AcT state and therefore, to enable OD, external writes inferred through data comparison should be recorded. This scheme basically needs a bit vector where bits get set to indicate change in data values. Other structures used in I-FSD such as flags to indicate coherence miss and false sharing as well as program counter counter that caused the miss are still needed to correctly perform OD.

2.3.7. Design Choices for Overlap Detection. We explore various design choices for OD shown in Figure 10. The most aggressive OD implementation (right column) maintains local state for all blocks in the coherent private caches. We call this *Total OD state*. As a result, it yields highest accuracy by capturing all local state needed to accurately perform OD. However, it is expensive to maintain local state bits for every cache block.

The next most aggressive OD implementation (second column) maintains local pool to track local state of a limited number of cache blocks that suffer coherence misses. We call this *Partial OD state*. This implementation does not contribute to any misclassification of coherence misses. However, a coherence miss may not be tracked when there are not enough entries in the local pool. The local pool entry is freed from the local pool after classifying the coherence miss. Since, at any given time, not all cache blocks are involved in sharing, a local pool (similar to global pool) works well for tracking local state in private caches. If the AcT involves maintaining partial state, then an entry is created in the local pool only if the global pool tracks the corresponding cache block.

The least aggressive OD implementation (left column) does not maintain local state corresponding to OD for any cache block. We call this *Zero OD state*. At the time of coherence miss, it simply observes whether the data address causing the miss suffers true or false sharing and classifies the miss at this point. This has the potential to overestimate false sharing misses in a program because early classification would ignore true sharing access on a block that might happen much after the point of coherence miss.

2.3.8. Non-primary Private Caches. Local state is relatively easy to update and check in a primary (L1) cache where all memory accesses are visible to the cache controller. In lower-level caches, only cache line addresses are visible. As a result, in systems where multiple levels of cache may be involved in coherence (e.g., with private L2 caches), information from a private non-primary (L2) cache should be passed to the primary cache (L1) when it suffers a cache miss. The L1 cache then keeps track of the flags that track coherence miss and false sharing for the L2 cache and forwards the value of these flags back to L2 when the block is replaced from the L1 cache. Although in this scenario we have caches directly communicating access related information, it only happens between different levels of private caches for the same core, and only on a coherence miss or eviction.

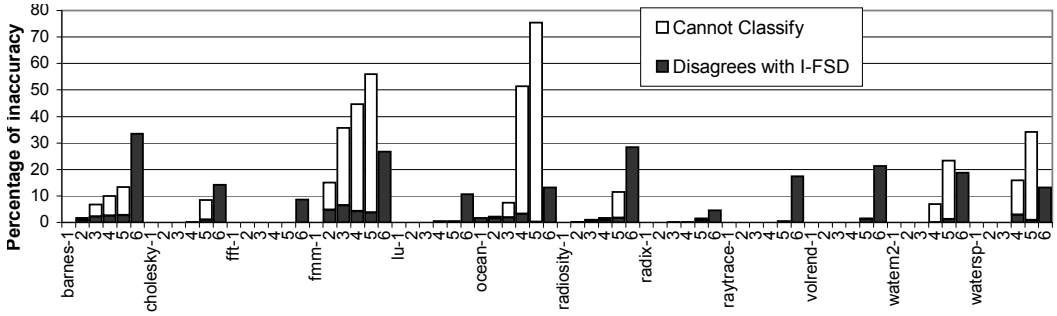
2.3.9. Other issues. The relatively simple mechanism to detect coherence misses described in Section 2.3.1 can err in two ways. First, on power-up or after a page fault, the state of a block's state is set to invalid, but the tag need not be initialized. This could sometimes result in a tag accidentally matching to that of a requested block. However, this would be quite rare and random enough that they do not attribute in significant numbers to any particular piece of code. Second, cache replacement policy could prioritize replacement of invalidated blocks, which can destroy the evidence of a coherence miss. For highly contended blocks involved in sharing patterns, there is little time for the block to be replaced between the invalidation and the subsequent miss, so replacement priority is unlikely to obscure enough coherence misses to hide a scalability problem. It should also be noted that the inaccuracy caused by replacement priority is very costly to avoid: it requires the cache to track the tags of blocks that *would be* in the cache were it not for replacement priority, e.g. using a duplicate set of tags.

For machines with shared bus interconnect and a shared last-level cache, the AcT state can easily be maintained in the shared cache. However, with the growing popularity of technologies such as HyperTransport [24], the interconnect could be high-speed, point-to-point communication links between the cores. In these cases, we maintain the global AcT state for a block in its home node or tile, along with existing coherence state that is already part of the directory. For all dirty block (cache blocks that had write accesses from the local core) replacements, the AcT state can be easily updated by piggybacking on existing writeback messages. However, for clean block (cache blocks that had read accesses only) replacements, the reader information is sent to the home node to update the AcT state using an extra message.

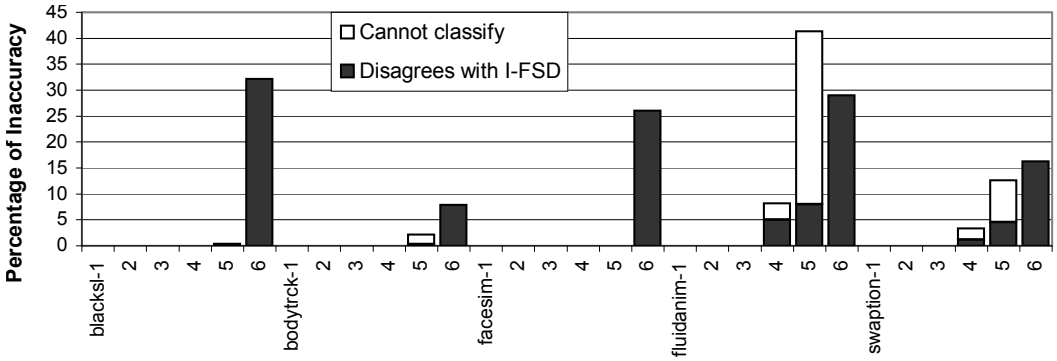
2.3.10. Evaluation. In our experiments, we use the same configuration used in Section 2.4.2 and measure coherence misses in large L2 private caches to maximize the number of coherence misses and hence study our approximate schemes. To examine how the approximations

in practical schemes affect their accuracy, we measure the percentage of coherence misses that are not correctly classified by the particular scheme. We identify two aspects of inaccuracy namely– a) the practical scheme can disagree with the I-FSD scheme, that is, practical scheme reports false sharing while I-FSD detects true sharing and vice versa and b) the practical scheme may not be able to classify true or false sharing due to lack of information from the AcT state or OD state. This results from maintaining partial state for AcT and OD.

Figures 11(a) and 11(b) show the accuracy results for a range of practical schemes that maintain total OD State and varying amounts of AcT state. Each benchmark is evaluated for six different schemes (from left to right) namely Near-Total AcT state (where AcT state is maintained for every cache block with a total of 256k entries), Partial AcT state with four different configurations where the global pool maintains has 16384(16k), 4096(4k), 1024 (1k), 64 entries respectively and finally Zero AcT state, where false and true sharing are inferred locally through data comparison.



(a) Splash-2 Benchmarks



(b) Parsec Benchmarks

Near-Total	Partial (16k)	Partial (4k)	Partial (1k)	Partial (64)	Zero AcT
4.36%	1.8%	1.79%	1.65%	1.57%	0.00%

(c) Area Overhead of AcT state as a percentage of on-die cache area. Total OD state adds 7.35% area overhead to on-die caches.

Figure 11. Accuracy versus cost trade-offs for practical schemes having total OD state and varying amounts of AcT state. For each benchmark, the six bars (from left to right labeled 1 through 6) represent Near-Total AcT state, Partial AcT state with 16k, 4k, 1k, 64 entries and Zero AcT state.

Near-Total AcT state achieves perfect accuracy in all of our benchmarks expect ocean where a small percentage ($<1\%$) of coherence miss classifications disagree with I-FSD. This inaccuracy results out of loss of AcT state that happens over very long time intervals. Partial AcT states with 16k and 4k entries capture enough accuracy with $<10\%$ of coherence misses being misclassified in almost all benchmarks except fmm and water-sp. Partial AcT states with 1k and 64 entries do not have not enough entries to accommodate all the blocks involved in sharing. From the attribution information, we find that a relatively few static instructions are involved in a large number of coherence misses. For instructions that infrequently suffer coherence misses, information about cache blocks they use are replaced from the global pool and hence introduce inaccuracy in these benchmarks. Our experiments show that a small percentage ($<5\%$) inaccuracy results from loss of state on certain reads and writes performed by certain cores. Even when the Partial AcT state has information for the cache block, an intervening replacement in the global pool might result in loss of access history about reads and writes performed by cores on that block. On the other hand, there are situations where inaccuracy results from unavailability of Partial AcT state. For such situations, there are two possible solutions– 1) For static instructions of interest (those that suffer coherence misses frequently), state about cache blocks they use can be tracked by better sampling techniques 2) Use Zero AcT (data comparison) technique as a fall-back mechanism to detect false sharing. Finally, Zero AcT with Total OD state incurs inaccuracy in the range of 4.5%(radix) to 33.5%(barnes). We find that the writer cores being unable to detect reads by other cores is a significant source of misclassification. The other major causes are silent writes in dynamic scheduling

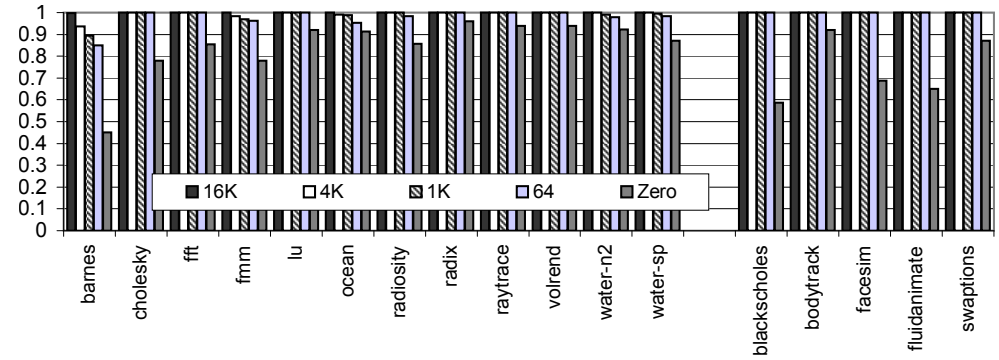


Figure 12. Correlation coefficient between number of false sharing misses suffered by static instructions in Partial AcT, Zero AcT schemes against Near-total AcT. Samples of static instructions are chosen by Partial AcT schemes.

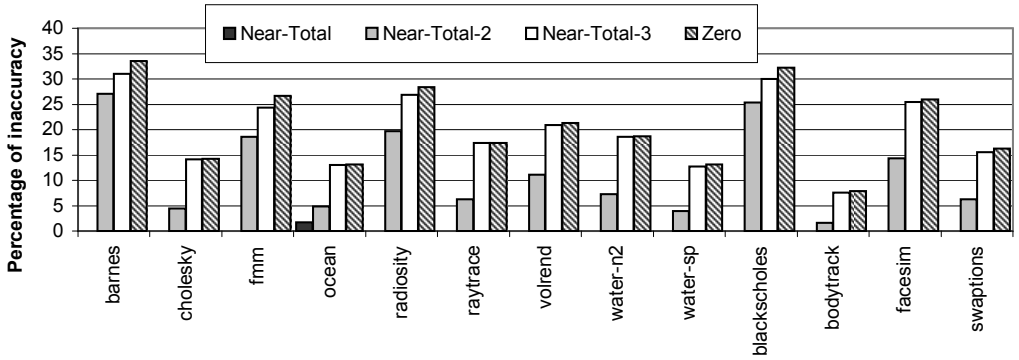


Figure 13. Percentage of inaccuracy for Zero AcT and different implementations of Near-Total AcT. For each benchmark, there are four bars- Near-Total, Near-Total-2 (does not consider reader information on the writer side), Near-Total-3 (does not consider silent writes and ignores reader information on the writer side) and Zero AcT.

code used by many PARSEC benchmarks as well Radiosity in Splash-2 benchmark suites. From attribution experiments, we find that top ten offending static instructions still match in many benchmarks, although they have differences in reporting the numbers of false and true sharing misses to those offenders.

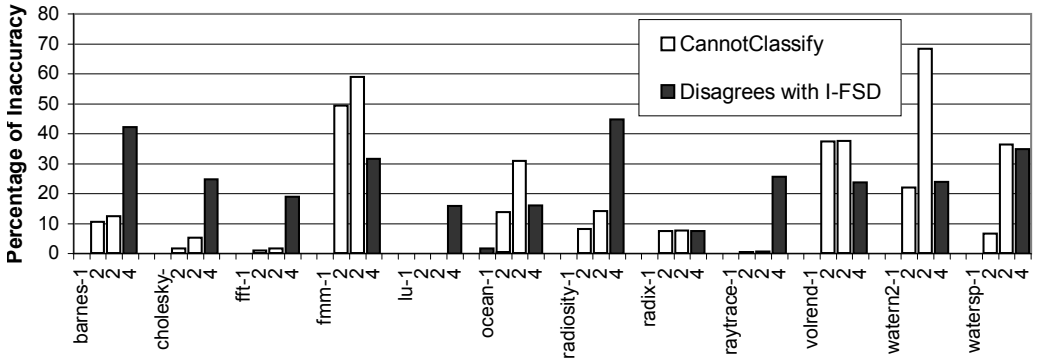
Figure 11(c) shows the area cost of our practical schemes with different AcT state as a percentage of total on-die cache area. The most aggressive Near-Total AcT scheme keeps the global AcT state for all words and maintains local OD state for the entire cache. As a result, it incurs up to 11.7% (4.36% for global state) area overhead compared to total area of the on-die caches used in our experiments. For all of the Partial AcT state schemes, the local state kept in each private cache dominates the cost. Thus reducing the amount of AcT state has little impact on area overhead. The least expensive Zero AcT scheme incurs 7.35% area overhead on on-die caches just to maintain the local OD state.

Overall, from the above experiments, we find that accuracy of partial AcT schemes is sensitive to the amount of AcT state, but area overhead is not very sensitive to the amount of AcT state, except for Zero AcT. As a result, a good design of a partial AcT scheme is one that maximizes accuracy by providing enough entries (e.g. 4k or more entries).

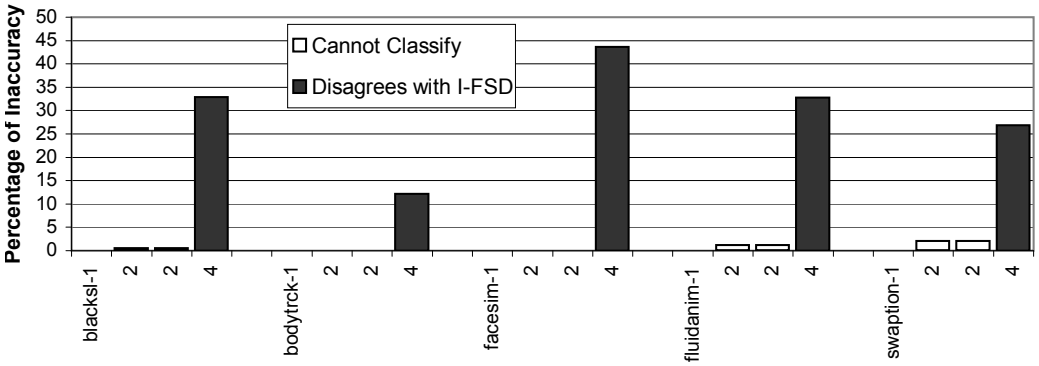
We perform additional experiments to provide insight into the various AcT schemes. The design points that perform Partial AcT classify coherence misses for cache blocks whose information is available in global state. Effectively, they sample a subset of coherence misses that are actually classified by Near-Total AcT. We study the correlation coefficient between the number of false sharing misses for the static instructions that are chosen as samples by the Partial AcT schemes. A correlation of +1 means that false sharing misses reported by Partial AcT are similar or proportional (equivalent modulo scaling) to the false sharing misses reported by Near-Total AcT for the chosen samples. A correlation of 0 means that they are uncorrelated (or independent of each other). Figure 12 shows the results of our experiments. We see that all benchmarks except barnes, fmm and ocean show high correlation ($>$0.99) even for 64 entries. This shows that even though the Partial AcT schemes perform classification only on a subset of static instructions causing coherence misses, we still get a very good accuracy from Partial AcT states. Zero AcT exhibits poor correlation against Near-total AcT with correlation coefficient as low as 0.45 in barnes benchmark.

We conduct experiments to study the differences between Near-Total AcT and Zero AcT and hence, verify our implementation. As first step, we modify Near-Total AcT to ignore reader information on the writer side. This is similar to Zero AcT not being able to detect reads on the writer side to declare true sharing. We call this implementation as Near-Total-2. We then modify Near-Total-2 to ignore silent writes. This is also similar to Zero AcT not being able to detect writes that do not change values of memory locations. We call this implementation as Near-Total-3. Figure 13 shows the result of our experiments where we measure the percentage of inaccuracy for Near-Total AcT, modified

implementations of Near-Total AcT and Zero AcT. In all benchmarks (except blackscholes), there is a noticeable increase in percentage of inaccuracy once we ignore reader information on the writer side for coherence miss classification (Near-Total-2). As we further ignore silent writes from consideration (Near-Total-3), the percentage of inaccuracy almost equals to Zero AcT in all benchmarks except barnes, fmm and blackscholes. On further investigation, we find that these benchmarks have a small percentage of idempotent writes (writes that change values and deposit the old values back between coherence misses). In these cases, Near-Total AcT would classify as true sharing while Zero AcT would detect false sharing because it does not detect change of data values.



(a) Splash-2 Benchmarks



(b) Parsec Benchmarks

Total OD	Partial(1k)	Partial(256)	Zero OD
7.35%	3.81%	3.66%	0.0%

(c) Area Overhead of OD state as a percentage of on-die cache area. Total global AcT state adds 4.36% area overhead to on-die caches.

Figure 14. Accuracy versus cost trade-offs for practical schemes having Near-Total AcT state and varying amounts of OD state. For each benchmark, the four bars (left to right labeled 1 through 4) represent Total OD state, Partial OD state with 1k, 256 entries and Zero OD state.

Figures 14(a) and 14(b) show the accuracy results for a range of practical FSD schemes that maintain Near-total AcT state and varying amounts of OD state. We evaluate for four different schemes for each benchmark (from left to right) – Total OD state (where OD state is maintained for every cache block with a total of 4k entries for each private cache), Partial OD state with two different configurations where the local pool maintains has 1024(1k) or 256 entries and finally Zero OD state (where false and true sharing are declared at the time of coherence miss).

Total OD state achieves the highest accuracy. However, Partial OD states exhibit slightly different behavior than Partial AcT states. From the figures, it should be noted that Partial OD states predominantly suffer from one type of inaccuracy with respect to I-FSD: inability to classify coherence misses when the local pool has too few entries. Our attribution experiments show that cache blocks that frequently suffer coherence misses are still accurately captured by Partial OD state configurations, even those with reduced number of entries. Finally, Zero OD with Near-Total AcT state incurs inaccuracy in the range of 7.5%(ocean) to 45%(radiosity). Early classification of coherence misses results in missing true sharing accesses that occur after the time of coherence miss (Section 2.2.1). Hence this scheme yields little value (accuracy) for the amount of global state maintained for each word. In contrast, at the opposite end of the design space, (Total OD state, Zero AcT state) yields higher accuracy at lower cost than this scheme (Section 2.3.6). This indicates that Zero OD schemes only make sense for extremely low cost designs i.e, only for combinations with Zero AcT.

Figure 14(c) shows the area cost incurred by practical schemes with different OD states as a percentage of total on-die cache area. Maintaining partial OD state for 1k entries incurs about 3.8% area overhead to on-die caches and for 256 entries, it costs about 3.66% additional cache area. For the Zero OD state, the area overhead of 4.36% is needed to keep global AcT state.

Overall, from the above experiments, accuracy drops marginally when moving from Total OD state to Partial OD states while we lose accuracy sharply when moving from Partial OD states to Zero OD states. Because the associated area overheads are similar for Zero OD and Partial OD, configurations with partial OD are preferable over Total OD ones (nearly similar accuracy for less cost).

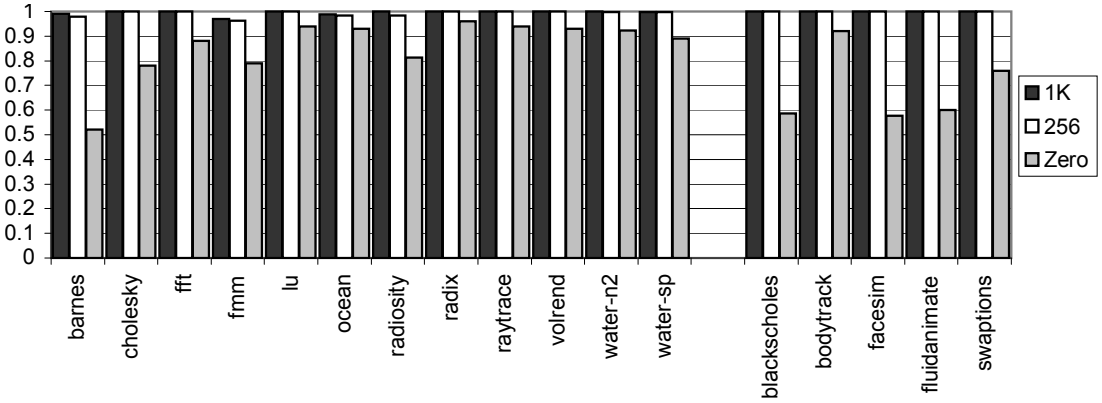


Figure 15. Correlation coefficient between number of false sharing misses suffered by static instructions in Partial OD, Zero OD schemes against Total OD. Samples of static instructions are chosen by Partial OD schemes.

Partial OD states sample a subset of coherence misses and classify them because of limited history information maintained by them. Effectively, these design points perform sampling and determine true and false sharing. We conduct experiments to study the correlation between the number of false sharing misses reported in both the Total OD and the corresponding Partial OD state for the static instructions that are chosen as samples by the Partial OD schemes. Figure 15 shows the results of our experiments. We see that all benchmarks except fmm show high correlation (>0.99) even for 256 entries. This shows that even though the Partial OD schemes perform classification only on a subset of static instructions causing coherence misses, we still get a very good accuracy for those samples from Partial OD states. Zero OD shows poor correlation against Total OD with correlation coefficient as low as 0.52 in barnes benchmark.

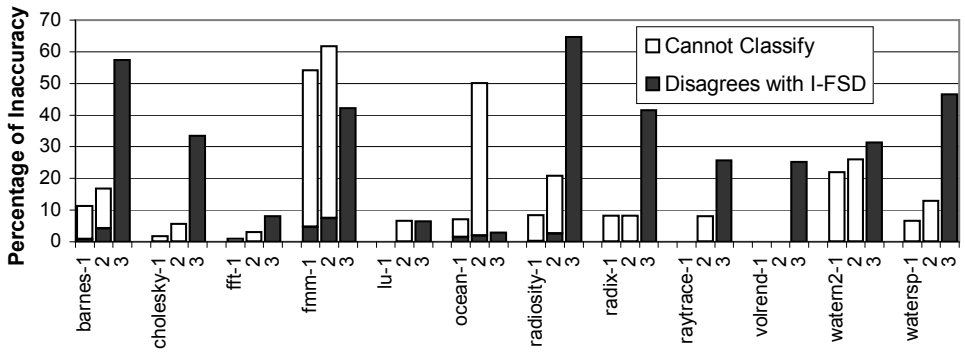
Even though, the preceding discussion appears to categorize the two phases of coherence miss classification namely *Access Tracking* and *Overlap Detection* into discrete design points, in reality, we get a continuous spectrum of design points along both the two axes. Partial State along both directions provides a continuum between maintaining zero state and full state. There are a wide range of possibilities in this design space. The configuration that is most appropriate for a user is to identify the region in the partial state that helps capture the entire picture for the programmer that would otherwise be obtained through maintaining total global and local states.

Figures 16(a) and 16(b) show the accuracy results for a range of FSD schemes that are relatively low cost and can be supported in real hardware. For each benchmark, we show three different schemes (from left to right) – i) (Partial(1k) OD, Partial (16k) AcT), ii) (Partial(256) OD, Partial(4k) AcT) and iii) (Zero OD, Zero AcT).

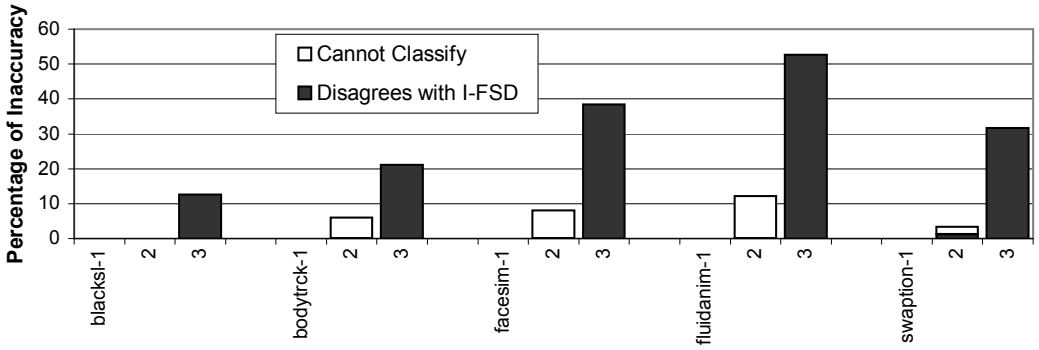
From prior experiments, we have seen that partial states offer reasonable accuracy while achieving lower costs. We pick partial AcT states that preserve relatively high accuracy (global pool having 16k and 4k entries) and evaluate them in combination with the partial OD state that has 1k and 256 entries in the local pool respectively. Our experiments show that this combination has similar accuracy to the Partial AcT states with 16k and 4k entries respectively in combination with Total OD. Benchmarks such as fmm, ocean and water-n2, which suffer additional inaccuracy from reduced number of OD entries. Finally the (Zero OD, Zero AcT) design point is the least accurate but is almost free in terms of cost. We find that this scheme has a highest misclassification percentage in many benchmarks with a maximum up to 65%(radiosity). This design point was used in Coherence Decoupling [23] to perform value speculation and hide cache miss latency due to false sharing. Although this design point is sufficient for speculation (where there are recovery mechanisms), it might produce misleading results for the programmer during performance debugging.

Figure 16(c) shows the area cost incurred by FSD schemes as a percentage of total on-die cache area. Partial schemes incur relatively modest area overheads of about 5.5% , a major portion (about 70%) of which is incurred by maintaining local pools for every private cache in our 64-core configuration. The (Zero OD, Zero AcT) scheme can be implemented with minimum hardware modification such as changing the cache controller to perform data comparison on a coherence miss. Hence, this scheme has near-zero cost. Overall, we find that partial schemes can offer relatively good accuracy compared to I-FSD with reasonably modest costs.

Schemes with partial OD and AcT state sample a subset of coherence misses and classify them because of limited history maintained both globally and locally. Effectively, these design points perform sampling and determine true and false sharing. We conduct experiments to study the correlation between the number of false sharing misses reported in both the (Total OD, Near-Total AcT) and the corresponding Partial state scheme for the static instructions that are chosen as samples by the Partial state schemes. Figure 17 shows the results of our experiments. We see that all benchmarks except barnes and fmm show high correlation (>0.99) even for 256 entries. This shows that even though the Partial state schemes perform classification only on a subset of static instructions causing coherence misses, we still get a very



(a) Splash-2 Benchmarks



(b) Parsec Benchmarks

Partial(1k) OD Partial(16k) AcT	Partial(256) OD Partial(4k) AcT	Zero OD Zero AcT
5.61%	5.45%	0.0%

(c) Area Overhead of OD and AcT states as a percentage of on-die cache area.

Figure 16. Accuracy versus cost tradeoffs for certain low-cost practical FSD schemes. For each benchmark, the three bars (left to right labeled 1 through 3) represent (Partial (1k) OD, Partial (16k) AcT), (Partial(256) OD, Partial(4k) AcT) and (Zero OD, Zero AcT) points in the design space.

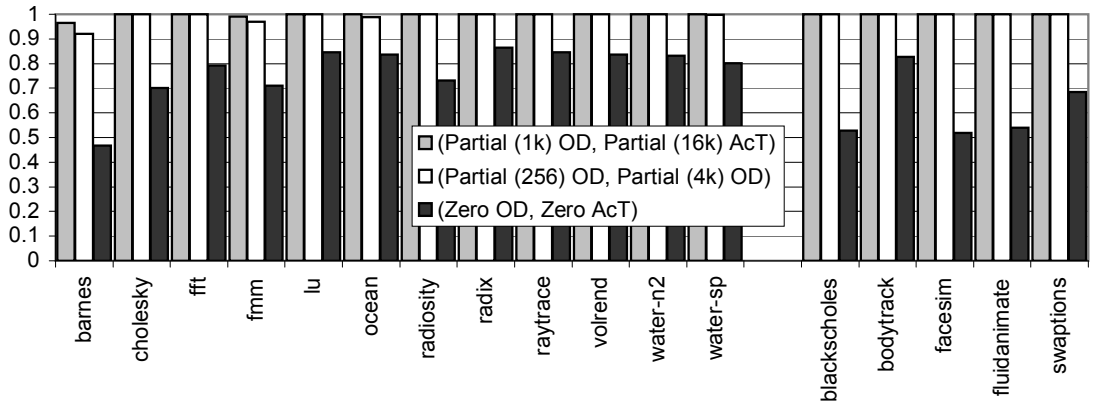


Figure 17. Correlation coefficient between number of false sharing misses suffered by static instructions in various schemes against (Total OD, Near-Total AcT). Samples of static instructions are chosen by Partial state schemes.

good accuracy for those samples from Partial states. (Zero OD, Zero AcT) shows poor correlation against (Total OD, Near-Total AcT) with correlation coefficient as low as 0.468 in barnes and 0.51 in facesim benchmarks.

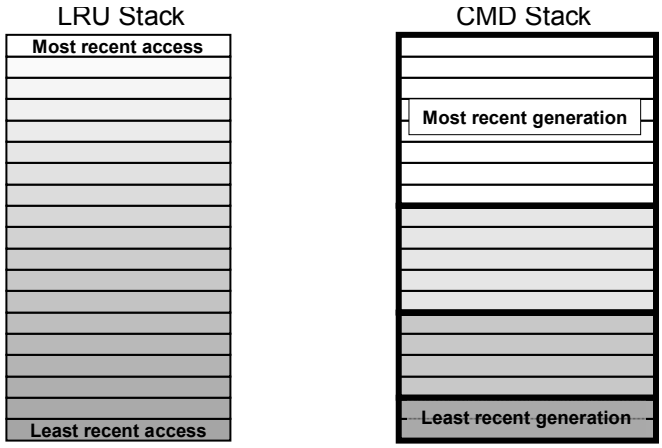


Figure 18. LRU stack (used by I-CMD) and CMD generational stack. The shading shows how recently a block was accessed. In the LRU stack, there is a total ordering among the blocks in the stack. In the CMD stack, no ordering information is maintained within a generation.

2.4 Classification of Replacement Misses

In this section, we first describe an ideal (off-line) scheme to identify conflict misses. We then present the disadvantages in implementing the ideal scheme and describe our practical conflict miss detector and its implementation. Our scheme can be used in any cache, regardless of its level in the memory hierarchy and whether it is private or shared.

2.4.1. Ideal Conflict Miss Detector (I-CMD). Replacement misses fall into two categories: capacity and conflict. A capacity miss occurs when the requested block was replaced from the cache because the cache did not have enough capacity to accommodate incoming cache blocks. Minimizing capacity misses involves a redesign of the algorithms used in an application (e.g., loop blocking) or upgrading to a processor with larger caches.

In contrast, a conflict miss happens when several blocks map into the same set in the cache and replace each other even when there is enough capacity left in the cache. When the number of these blocks exceeds the cache associativity, a block, A, will be evicted even though better candidates for eviction may exist in the cache in other sets. If A is accessed again before those better candidates are replaced, that access is a conflict miss. That is, a fully associative cache of the same capacity would have kept A in the cache and not incurred the miss. Minimizing conflict misses involves rearranging data so that blocks frequently accessed together map to different sets. This can be done through padding or changing the alignment of data structures.

For caches that use a least recently used (LRU) replacement policy, the standard off-line technique to detect conflict misses is to maintain an LRU stack with N entries that tracks the blocks that would be present in a fully associative N-block cache. On each access, the block being accessed is placed on the top of the stack. If this block was already present in the stack, it is also deleted from its prior location in the stack. A miss is classified as a conflict miss if, prior to the stack update, the block is found on this N-entry stack. An LRU stack is usually used for off-line analysis of memory accesses [22, 37]. However, implementing this off-line algorithm in hardware for on-the-fly classification would be prohibitively expensive in terms of performance overhead, area cost and power budget.

2.4.2. Practical Conflict Miss Detector. To keep cost low, we propose a new, less complex Conflict Miss Detection scheme that approximates the access-recency information provided by an LRU stack used by I-CMD. In particular, we use a scheme that maintains a number of *generations* that are ordered by age. Each generation consists of a set of blocks, and all blocks in a younger generation have been accessed more recently than any block in an older generation. This means that the blocks in the youngest generation are the blocks that would be at the top of the LRU stack, the next (older) generation corresponds to the next group on the LRU stack, etc. Figure 18 shows an example using four generations.

Unlike an LRU stack, our generational scheme does not track the recency of access ordering within a generation. For example, if we miss on a block that is found in the youngest generation, and if that youngest generation currently contains 8 blocks, we only know that the requested block is among the 8 most recently accessed blocks. In contrast, an LRU stack would provide precise information—if the block is found in the second position counting from the top of the stack, then that is the second most recently accessed block.

In our scheme, we start with an empty youngest generation and add cache blocks to it as they are accessed until the number of blocks in this generation reaches a threshold number T. After this, we begin a new youngest generation, aging the previous youngest generation and all generations older than it. Because the state used to track generation membership is finite, we only maintain up to K generations. Once we have created the first K generations, the oldest generation is erased and its resources reused to create the new youngest generation. This corresponds to the removal of the bottom entry on the LRU stack when a new entry is added at the top and all other entries are pushed down one place. However, in our generational scheme we remove an entire generation at a time, which results in imprecise classification at the boundary between capacity and conflict misses.

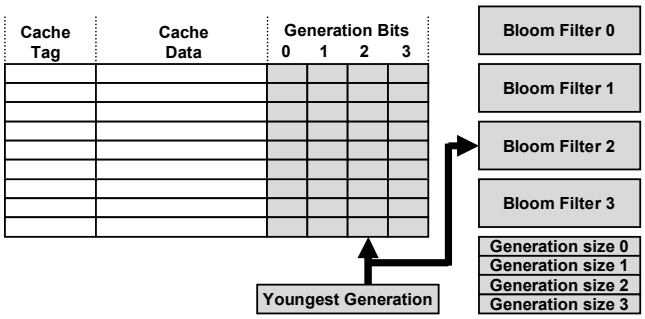


Figure 19. Shaded areas show the hardware added to implement our Conflict Miss Detector (CMD). A 4-generation CMD is shown, at a time when Gen 2 is currently the youngest one.

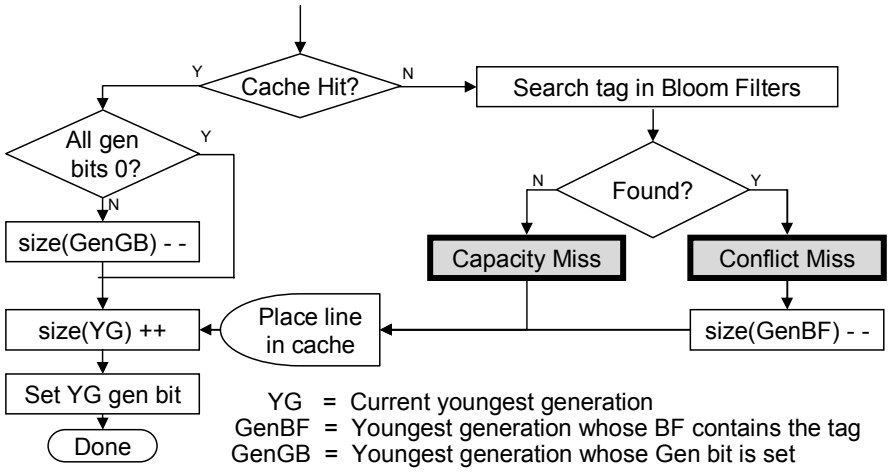


Figure 20. CMD implementation flowchart.

2.4.3. Implementation. Figure 19 shows the hardware components added to a cache in our proposed implementation of CMD and Figure 20 shows how these components are used to detect conflict misses. First, we need to keep track of the current youngest generation since the generations are reused similar to circular buffers. For instance, in Figure 19, the ordering of the generations is 2 (youngest), 1, 0, and 3 (oldest). Second, we maintain an array of counters that tracks the size of each generation. These counters and the youngest generation register contribute little to the overall cost of the scheme. Finally, the membership of a generation is split between two structures. Blocks that are present in the cache are tracked using *generation bits*. Blocks that are not present in the cache are tracked using a Bloom filter [6] for each generation.

Generation bits are added to each cache entry. Each bit corresponds to one generation. To put the block into the youngest generation, we simply set the corresponding bit. To determine which generation the block currently belongs to, we determine the first bit that is set (in order of generation age). To clear the state of the oldest generation, we flash-clear all bits that correspond to that generation (and all bits in the corresponding Bloom filter).

When the number of generations is relatively large (8 or more), an in-cache representation that is more compact than generation bits may be preferable. One approach is to record the actual generation number (GN) for each cache block. To put the block into the youngest generation, the current generation number is written into the block's GN. Generation membership of the block can easily be determined by reading its GN. However, to clear a generation we must find all blocks in that generation and change their GNs to a special "no-generation" number. To represent the "no-generation" number, we can add a Generation Valid (GV) bit, which is set only for blocks that belong to some generation. When a new generation is created, we must clear the GV bits for blocks whose generation number matches the new current number. To avoid sweeping through the cache for this purpose, we use a sliding window scheme that adds another bit to the GN field itself. When the current generation number changes, if the most significant bit (MSB) changes, all blocks whose GN's MSB matches it have their GV cleared. In other words, when the MSB flips, the oldest generation numbers are changed to "no-generation".

Note that this generation-number scheme uses $2+log_2(K)$ bits per in-cache block, while the more straightforward generation-bits scheme uses K bits per in-cache block. This means that the generation-bits scheme is preferable when $K \leq 4$.

Each generation uses a Bloom filter to keep track of the blocks that belong to that generation but are not present in the cache. With an N-line cache and K generations, there are K Bloom filters. Each is a three-hash Bloom filter with 4*N/K bits of state, which allows each filter to track all the blocks in its generation with a very low false-positive rate [6]. It is interesting to note that the total size of all the Bloom

filters for a given cache is independent of the number of generations (K filters, each has 4*N/K bits). Also, our Bloom filters only need to keep track of *replaced* blocks in a generation, not the entire generation. As a result, we could use considerably fewer than 4N bits and still maintain excellent accuracy [20]. Finally, it should be noted that these Bloom filters are accessed only on cache misses, so they do not affect hit times, can be single-ported, and can be implemented with cheaper and slower circuitry than the corresponding cache.

Individual entries cannot be deleted from Bloom filters. However, in CMD, an entry in a Bloom filter should ideally be deleted from an older generation's Bloom filter when it is "stolen" by the youngest generation. Instead, we note that older generations are always deleted before younger generations, so it is safe to leave deleted entries in the Bloom filters. The actual generation of a block is now the youngest generation in which that block can be found.

When a cache block is evicted, its generation membership is recorded in the corresponding Bloom filter. When a block is brought into the cache, it will be tracked by the generation bits even if it was already in a Bloom filter. As a result, for in-cache blocks we only need to check and update generation bits, and Bloom filters are only checked on cache misses and updated on cache replacements.

Our CMD scheme should have little effect on the cache hit or miss latency. For a cache hit, the generation bits can be read while the tags are being checked. If the youngest-generation bit is not set already, it needs to be set. This can be accomplished using the same approach used to change the state of the block from clean to dirty on a write. The update of generation size counters, if it is needed, is not on the critical path of the cache access and can be performed in the next cycle.

On a cache miss, the Bloom filter lookup and the CMD classification of the miss can begin as soon as the cache miss is detected, and can easily be completed before the requested data arrives from memory or a lower-level cache.

A final consideration is how to attribute capacity and conflict misses to the instructions that trigger them. Performance debugging in existing processors is typically supported by including several performance counter registers. Each of these registers is typically associated with a control register that determines which event will be counted (e.g., committed instructions, cache accesses, L1 cache misses, etc.). To attribute events to a particular instruction, the counter is typically initialized to some value, and counts down each time the event occurs. The hardware raises an exception when the count reaches zero, and the program counter value is recorded by a software tool. This sampling approach allows the software tool to provide a programmer an approximation of which instructions triggered the most events of interest, while incurring little execution time overhead. We rely on this conventional mechanism for capacity and conflict miss attribution. However, our scheme can be even more beneficial with more advanced profiling mechanisms such as ProfileMe [13] and stratified sampling [45].

2.4.4. Evaluation. We evaluate our practical generational CMD against off-line I-CMD. We use SESC [43], a cycle-accurate, execution driven simulator. We model 64-core chip multiprocessor. Each core is a 2.93GHz, four-issue, out-of-order processor with a 32KB, 4-way set-associative private L1 cache (2 cycle hit and 1 cycle miss latencies), 512KB, 16-way set-associative private L2 cache (10 cycle hit and 4 cycle miss latencies). All cores share an 8MB, 32-way L3 cache, and the MESI protocol is used to keep L1 caches coherent. The block size is 64 bytes in all caches. Since conflict misses are more likely in caches with lower associativity [21], we use L1 caches to evaluate effectiveness of our CMD mechanism. New entries are only added to the youngest generation. So the size of a generation will never be more than the threshold T. A seemingly obvious choice is to set T to N/K, where N is the number of blocks in the cache. We conduct two sets of experiments with L1 caches implementing two different cache block replacement policies – Least Recent Used (LRU) and a slightly modified version of Clock-based Not-Recently-Used (NRU) algorithm used in SUN UltraSparc V2 [47]. LRU replacement policy maintains age information for every cache block that tracks the recency of access for that block. LRU chooses the least recently used block (or the block with maximum age) to be replaced with the incoming block. In the Clock-based algorithm, each cache block maintains a 'used' bit that gets set when the block is accessed or initially fetched from the memory. In addition, each block also has 'allocate' bit, which gets set when allocated and is cleared when the block is filled with data. Each cache set has a rotating replacement pointer, which is the starting point to find the *way* to replace. On a miss, the algorithm looks for the first cache block that has both 'used' and 'allocate' bit clear, starting with the *way* pointed by the replacement pointer. If all block have 'used' and 'allocate' bits clear, all 'used' bits are cleared and the scan is repeated. The replacement pointer is then rotated forward one *way*. Both LRU and Clock-based algorithms are used to contrast how well our practical scheme captures conflict misses compared to I-CMD (that uses a fully-associative LRU stack to determine conflict misses).

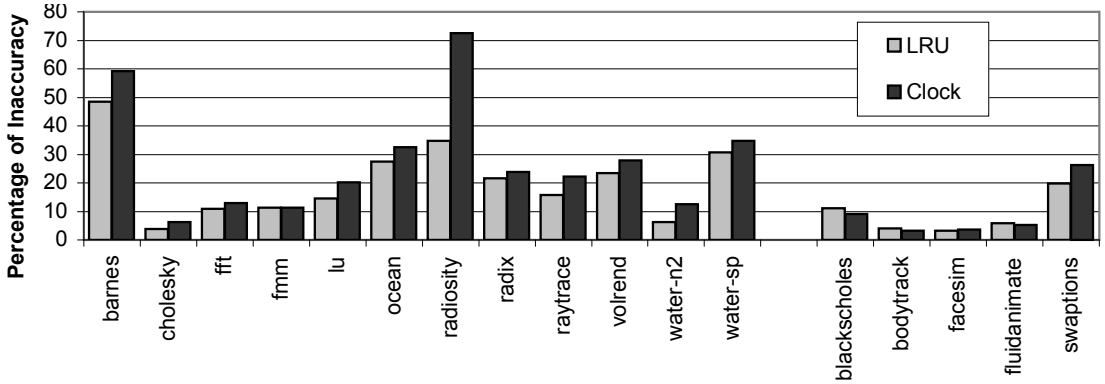


Figure 21. Percentage of inaccuracy for generational CMD on caches with LRU and Clock-based Replacement policies.

In our experiments, we measure the percentage of replacement misses that are not correctly classified by the practical CMD scheme. We call this as *percentage of inaccuracy*. Figure 21 shows the results of our experiments with number of generations, $K = 4$. Each benchmark has two bars which show the percentage of inaccuracy for LRU and Clock-based replacement policies. For LRU replacement policy, all benchmarks except barnes, radiosity and swaptions have inaccuracy less than 20%. On further analysis, we find that in these benchmarks, a significant portion of the conflict misses occur near the bottom of LRU stack in I-CMD scheme. Such misses are incorrectly classified as capacity misses by our scheme due to loss of information when the older generations are cleared. Even for benchmarks with significant (>30%) inaccuracy, our attribution experiments show that the static instructions that are identified as top ten offenders in I-CMD scheme mostly match with the top ten offenders in the practical scheme. For performance debugging efforts, this information is useful to help programmers fix their code that are hot-spots for conflict misses. Inaccuracy in practical CMD for clock-based replacement policy differs from LRU because of additional "randomness" while choosing blocks for replacement in comparison to LRU. The set of blocks that suffer replacement misses are quite different from LRU. Therefore, the percentage of inaccuracy depends on whether the classification of replacement misses for the blocks chosen by Clock-based policy concur in I-CMD and our practical scheme. In some benchmarks such as barnes and radiosity, there is significantly higher inaccuracy (compared to LRU replacement). In others, the percentage of inaccuracy for both replacement policies differ from each other by <5%. Our attribution experiments show that static instructions that are top offenders in the I-CMD and practical scheme largely match (except for a few cases in barnes and radiosity) and thus, our practical generational CMD scheme helps programmers reach similar conclusions regarding conflict misses.

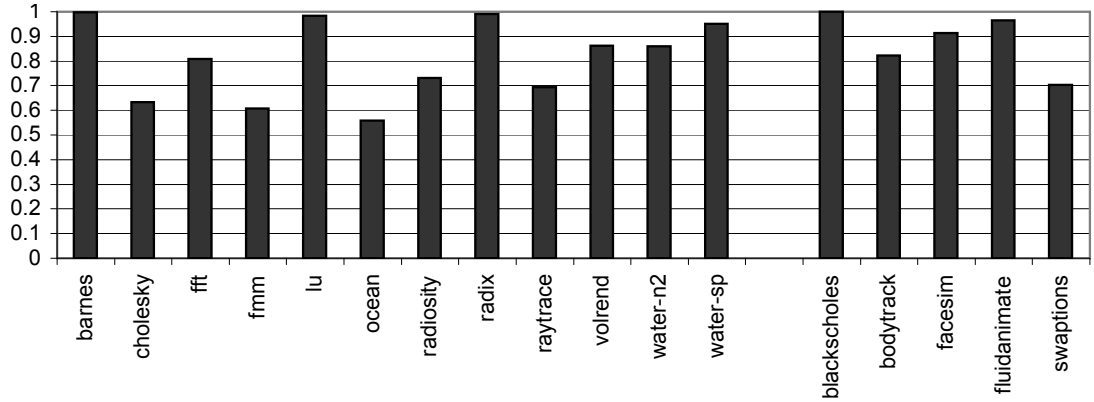


Figure 22. Correlation coefficient between conflict misses suffered by all static instructions in caches with LRU and Clock Replacement policies.

We note that LRU and Clock-based algorithm are two separate replacement policies. As a result, the set of static instructions that suffer replacement misses in a cache with LRU replacement policy will not be the same as those observed in a cache with Clock-based replacement. By using the LRU stack (I-CMD), we identify all the program counters (static instructions) that suffer conflict misses in both policies and measure the correlation coefficient (also known as Pearson product-moment correlation coefficient [12]) between the corresponding sets of conflicts. This is done to study how closely the Clock-based algorithm approximates the LRU replacement policy in causing conflict misses in caches. A correlation coefficient of +1 means strong correlation and that the conflict-based replacements that happen in both policies are similar or proportional to each other (equivalent modulo scaling). On the other hand, a correlation coefficient of 0 means that they are uncorrelated (independent of each other). Figure 22 shows the result of our experiments. We find that, in certain benchmarks like barnes, lu, radix, water-sp, blackscholes and fluidanimate, we observe a correlation coefficient of 0.95 or above. In other benchmarks, the correlation is not very strong (less than 0.9). From these results, we infer that programmers' intuitive understanding of conflict misses (defined by Hill et al. [22]) is already somewhat blurred by pseudo-LRU replacement policies such as Clock-based algorithm. With this in mind, we believe that our scheme can serve as a good tool for performance debugging purposes even though it is not as precise as I-CMD, because it can still identify static instructions responsible for the majority of conflict misses.

One of the main sources of inaccuracy in our practical CMD scheme stems from the threshold $T = N/K$ being sometimes too small; when an entry being added to the youngest generation is already present in an older generation, it is effectively deleted from that older generation. As a result, older generations become smaller over time due to this "stealing" by the youngest generation. Therefore, the number of blocks in all K generations combined could be considerably less than N. This results in numerous conflict misses (to blocks that would be in the lower part of an LRU stack) being misclassified as capacity misses. We call such conflict misses as "near-capacity" misses because these misses would be capacity misses in a slightly smaller cache. Near-capacity conflict misses by their very nature occur on blocks that stay in the cache for some time; in contrast, conflict misses on very recently accessed blocks occur after a short time and tend to be a result of thrashing. Thus, near-capacity misses typically have a much smaller impact on performance than other conflict misses.

Figure 23 shows the breakdown of replacement misses. For each benchmark, there are two bars – the first bar shows the breakdown in I-CMD scheme and the second bar shows the breakdown in the practical scheme. Conflict misses in I-CMD are broken down as near-capacity and conflict, that is, misses on blocks that are in the top (most recent) 75% of the LRU stack are still shown as "conflict," but misses on blocks

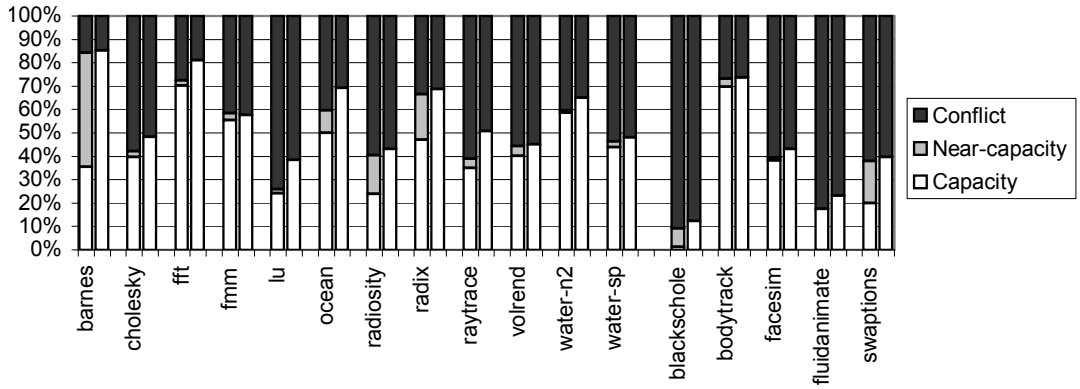


Figure 23. Breakdown of replacement misses. Each benchmark has two bars- the left bar shows the classification according to I-CMD and right bar shows the classification according to our practical CMD.

that are in the bottom (least recent) 25% of the LRU stack are shown as "Near-Capacity." We see that our practical generational CMD scheme performs very close to the I-CMD (offline LRU scheme) for more-recent conflict misses, but misclassifies Near-Capacity conflict misses.

Unfortunately, lack of information about ordering within a generation prevents our scheme from re-balancing generations by moving the oldest block in a younger generation into the next older generation. This means that the only way of controlling generation sizes is the threshold T that determines the final size of the youngest generation. However, if the threshold is chosen to be significantly larger than N/K, insufficient "stealing" by the youngest generation may result in more than N blocks being represented and, consequently, some capacity misses (to blocks that would be just below the N-th place in the LRU stack) would be classified as conflict misses.

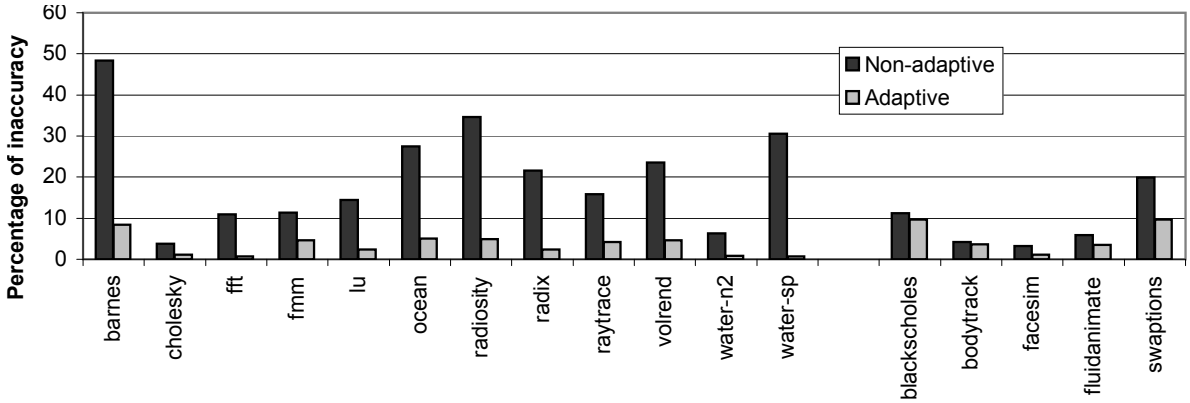


Figure 24. Percentage of inaccuracy in our practical CMD when thresholds in generations are non-adaptive and adaptive.

To manage the trade-off between these misclassifications, there are two possible solutions – i) adaptive threshold, and ii) increasing the number of generations. The adaptive threshold scheme determines the threshold T based on the number of distinct accesses tracked by all generations. Whenever we are about to create a new generation, we determine the total number of blocks represented in all generations. If this number is less than N, the threshold is increased to enlarge future generations. Conversely, if all generations together have more than N blocks, the threshold is lowered to shrink future generations. Finally, we avoid misclassifications of capacity misses by ignoring old generations which are not entirely above the Nth position on an LRU stack. In other words, if the sum of generation sizes for the L youngest generations is larger than N, then generation L and older are ignored because a miss that finds its block in one of those generations may be a capacity miss.

Figure 24 shows the result of our experiments. For each benchmark, the percentage of inaccuracy is studied for both non-adaptive (where T is fixed at N/K) and adaptive schemes. We find that through adaptive threshold, we significantly reduce the percentage of inaccuracy to $<$10% in all benchmarks (maximum of 9.7% in swaptions). The adaptive scheme is conservative in that it never misclassifies capacity misses as conflict misses; however, it may still misclassify some "near-capacity" conflict misses to blocks near the end of the (conceptual) LRU stack. Also, near-capacity misses that remain even with adaptive thresholds are less amenable to being avoided through padding and alignment—such fixes may simply convert these misses into true capacity misses. Therefore, we consider misclassification of some near-capacity misses acceptable for performance debugging purposes.

A second solution to minimizing inaccuracy is to increase the number of generations used to track conflict misses. When each generation holds N/K entries, with larger K, lesser number of entries (information about certain cache block accesses) are lost when erasing the oldest generation. Figure 25 shows the results of our experiments using 4, 8 and 16 generations. Across all benchmarks, the percentage of

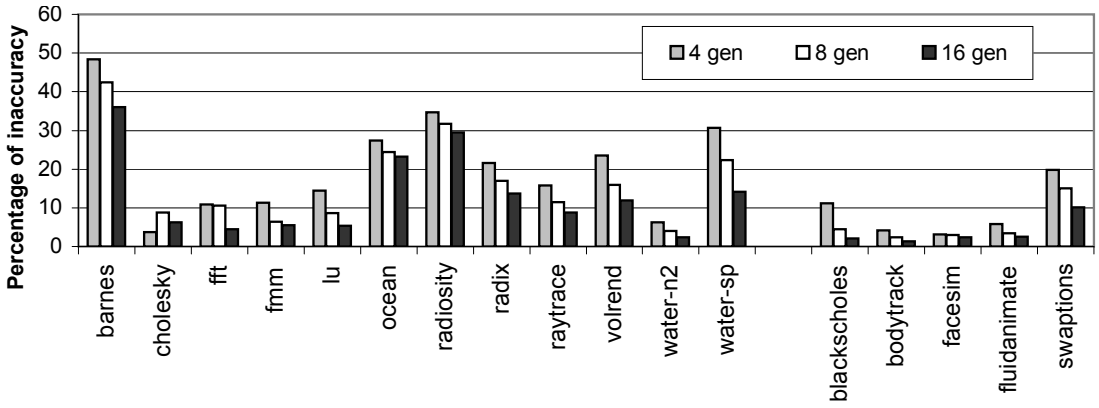


Figure 25. Accuracy of classification for 4, 8 and 16 generations.

Table 2. Latency, area, and energy overheads.

Private 32KB L1			
	L1 Access Time	**Total L1 Area**	**L1 Power**
Ideal-CMD	18.48%	111.65%	86.46%
Practical CMD	1.48%	3.89%	0.70%
Private 512KB L2			
	L2 Access Time	**Total L2 Area**	**L2 Power**
Ideal-CMD	101.87%	75.37%	310.28%
Practical CMD	1.58%	2.98%	1.01%

inaccuracy drop gradually as we increase the number of generations. However, even with 16 generations, the inaccuracy is still >30% in certain benchmarks like barnes and radiosity. This is because, a significant portion of conflict misses occur near the very bottom of LRU stack in the I-CMD scheme which are misclassified in the practical scheme.

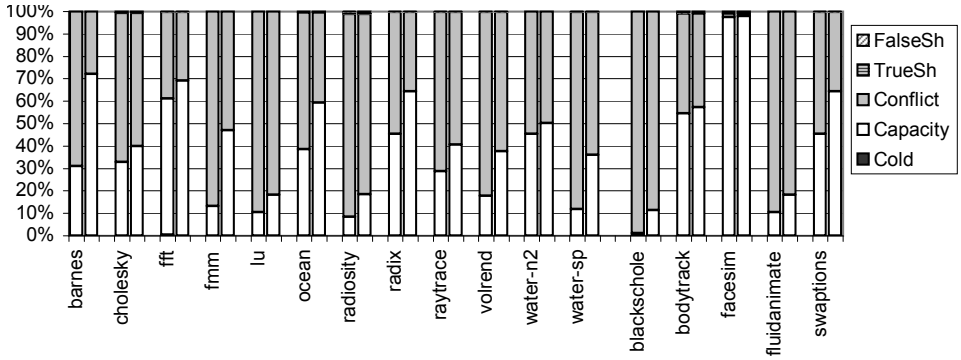
Table 2 shows the latency, area, and per-access energy overheads of both the offline I-CMD and our practical CMD schemes. The ideal scheme must update the LRU stack on every access. Since it is essentially a tag array for a fully associative cache, it has an access latency that is more than twice that of the baseline cache and requires up to 3x extra energy per access in larger L2 caches. Our proposed CMD mechanism with $K = 4$ generations uses 4 Bloom filters, each with $4 * N$ bits where N is the number of cache lines. It also uses 4 generation bits per line in the cache's meta-data array. The Bloom filters are used only on cache misses, so they do not affect cache latency and have minimal impact (which we account for) on power. The extra bits in the cache's meta-data array affect cache latency slightly, but this 1.5% increase in latency is unlikely to result in requiring an extra cycle for a cache hit. The overall energy impact of our CMD mechanism is minimal—it increases per-access energy by about 1%. Finally, our CMD mechanism uses less than 4% extra area in total, relative to the original area of the cache.

2.5 Comprehensive classification

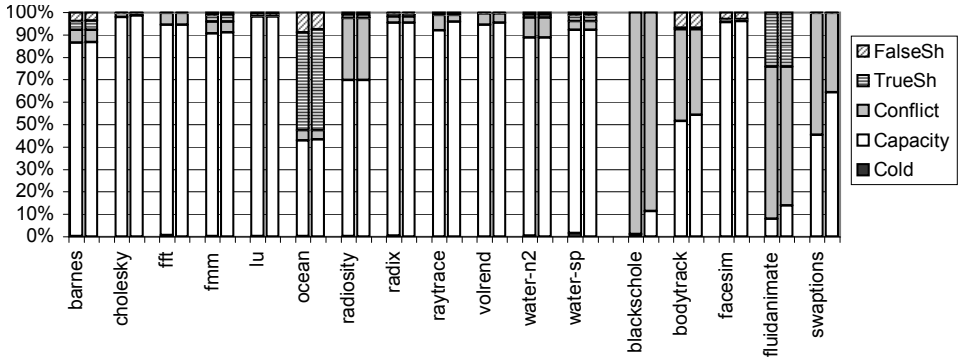
In this section, we perform comprehensive classification of cache misses using ideal and practical schemes described in Section 2.4 and 2.3. We use I-CMD and I-FSD schemes to show how ideal off-line schemes would classify cache misses and compare them against practical schemes with reasonable cost-accuracy trade-offs. We use practical CMD with 4 generations for classifying replacement misses and (Zero AcT, Total OD) for classifying coherence misses.

We perform experiments to study the cache miss classification both in 32 KB, 4-way private L1 caches (results shown in Figure 26(a)) and 512 KB, 16-way private L2 caches (results shown in Figure 26(b)). Each benchmark has two bars, the first bar shows the cache miss breakdown according to ideal offline schemes and the second bar shows the breakdown of cache misses in the practical schemes. In L1 caches, coherence misses form a negligible (<1%) fraction of the overall cache misses. Among the replacement misses, misclassification of conflict misses as capacity misses arises from "near-capacity" conflict misses (See Section 2.4.2). In L2 caches, coherence misses form a significant portion in certain benchmarks like barnes (8%), ocean (52%) and fluidanimate (25%). Conflict misses represent less than 10% of overall cache misses except in a few benchmarks like radiosity, blackscholes, bodytrack, fluidanimate and swaptions. This phenomenon is largely because of increased associativity in L2 compared to L1 caches [21]. Finally, cold misses represent negligible (<0.1%) of cache misses in both L1 and L2 caches.

Figure 27 shows the effect of different types of cache misses on pipeline stalls. When a load or store instruction occupies the head of Re-Order Buffer (ROB) and is unable to retire because of a cache miss, we attribute the pipeline stall to that cache miss. Our experiments show that capacity misses incur the highest average with 8.5 stall cycles per miss and true sharing misses have the least average with 7.14 cycles per miss. However, we note that all types of cache misses incur similar penalties in terms of pipeline stalls. This confirms our intuition



(a) private L1 cache



(b) private L2 cache

Figure 26. Cache miss classification for smaller (32 KB) L1 and larger (256 KB) L2 caches.

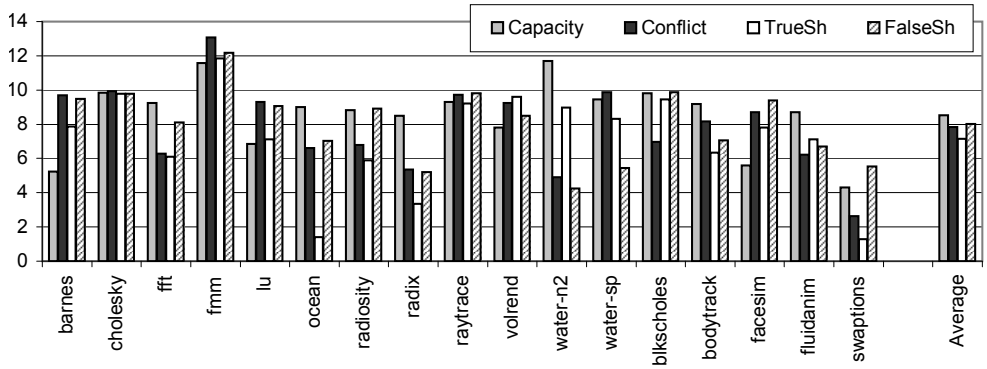


Figure 27. Average per-miss pipeline stall cycles incurred by different types of cache misses.

that the symptoms of cache misses are often similar, whereas the fixes needed to address them can be very different, and often, depend on the type of the cache miss.

2.6 Related Work

The classification of uniprocessor cache misses into compulsory (cold), capacity, and conflict was first defined by Hill [22], and the stack algorithm for simulating a fully-associative cache is first described by Mattson et al. [37]. True and false sharing misses have been defined by Torrellas et al. [50], Eggers et al. [17], and Dubois et al. [15]. Each of them also describe an off-line classification algorithm. In contrast to these schemes and definitions, the mechanisms we describe are designed to be implemented in real hardware for integration with existing on-line performance debugging infrastructures.

Coherence Decoupling [23] speculatively reads data values from invalid cache lines to hide the latency of a cache miss caused by false sharing. It then uses the incoming (coherent) data values to verify successful value speculation. If the values differ, recovery action is triggered. The primary aim of this work was to hide latency of load instructions that read unchanged values. One of our design points (Zero OD, Zero AcT) in Figure 10 to classify coherence misses is similar to this implementation.

Numerous research proposals have been made for improving the performance counter infrastructure [45], attribution of performance-related events to particular instructions [13], and for sampling and processing of profiling data [2, 38, 39, 54]. Our cache miss classification mechanisms are synergistic with improvements in performance counters, sampling, and profiling infrastructure. Our mechanisms provide on-line identification of specific types of cache misses, and this identification can be used to drive performance counters, attributed to particular instructions, and processed further to gain more insight into program behavior and performance. The better the profiling infrastructure, the more beneficial the results of our classification are to the programmer. Conversely, our scheme enhances the value of a profiling infrastructure by providing additional event types that can be profiled.

Our CMD scheme relies on an approximation of the LRU stack algorithm. A similar generational approach has been used by Kim et al. [30] to perform efficient cache simulations and, more recently, by Zhou et al. [53] to dynamically track working set sizes in order to help allocate pages of physical memory. Our generational scheme is designed for conflict miss detection in caches, so its state is updated and looked up much more often that the working set mechanism in [53]. Because of this consideration, we use Bloom filters and in-cache generation numbers to track members of each generation, rather than hardware-maintained link-lists. We believe that our Bloom-filter based approach could be used to simplify and speed up the implementation of Zhou et al.'s working set scheme.

Collins et al. [11] propose a hardware scheme, the CMT, that identifies conflict misses by storing the tag of the last evicted block from each set. On a cache miss, if the miss is to the most recent victim for the corresponding set, the miss is classified as a conflict miss. We expect this scheme to be less accurate than ours for at least two reasons. First, since the CMT only remembers the last evicted block from a set, when a large number of lines compete for a set, many of these misses will be incorrectly identified as capacity misses. Also, since the CMT does not track recency information across different sets, infrequently accessed sets may suffer capacity misses that are classified as conflict misses. It should be noted that the CMT is primarily directed towards helping prefetchers and victim caches, where performance improvement can be achieved with moderately accurate miss classification. In contrast, our CMD scheme provides cache miss classification to drive performance counters and hardware-assisted profiling, where misclassification of capacity misses as conflict misses can mislead the programmer into fruitless data-padding optimizations.

2.7 Summary

Performance debugging of parallel applications is extremely challenging, but achieving good parallel performance is critical to justify the additional expense of parallel architectures. One particularly challenging performance debugging problem is determining the causes and sources of cache misses. This is especially true for parallel applications since they potentially suffer misses from sources not present for uniprocessor applications (e.g., false sharing and destructive sharing).

We proposed new schemes to perform on-the-fly classification of cache misses in hardware: a Conflict Miss Detector (CMD) that classifies replacement misses into capacity and conflict misses, and a False Sharing Detector (FSD) that classifies coherence misses into false and true sharing misses. We evaluated our scheme on SPLASH-2 and Parsec-1.0 benchmarks and find that it is provides a similar picture of cache miss-related performance problems to previously proposed off-line schemes. Combined with existing performance counter mechanisms or with more advanced profiling mechanisms, our scheme also pinpoints the code responsible for each type of cache misses in the application.

3 LIME - A Load Imbalance Reporting Tool

Load imbalance is one of the key scalability limiters in parallel applications. Ideally, a parallel application assigns an equal amount of work to all cores, keeping all of them busy for the entire application. Load imbalance occurs when some cores run out of work and must wait for the remaining cores to finish their work.

Load imbalance is relatively easy to detect—we can watch for threads waiting at the end of a parallel section (i.e., at a barrier) or at a thread-join point. However, it is much more difficult to diagnose the cause of load imbalance with sufficient precision to help programmers decide what changes to make to the application to reduce the imbalance.

The cause of load imbalance can be hard to diagnose because there are a variety of candidates. For instance, load imbalance can be caused by assigning an unfair proportion of tasks to a thread, or by assigning too many long tasks to the same thread—both of these manifest as control flow differences between threads. Diagnosing causes for imbalance becomes even harder when it occurs due to interactions between the application and the underlying hardware (e.g., threads having different numbers of cache misses). Such causes cannot be easily detected through code inspection or static analysis.

Reducing load imbalance has long been an active research topic. The most common approach to reducing load imbalance is to use dynamic task scheduling, such as that provided with OpenMP [41] and TBB [25]. Dynamic task scheduling involves partitioning the parallel work into many more tasks than threads, and using a run-time system to assign tasks to threads on-demand. Dynamic scheduling significantly reduces load imbalance, but introduces significant runtime overheads, both from executing scheduling code and from the loss of cache locality among tasks. These overheads increase with the number of cores/threads, and can dominate performance [31].

Because load imbalance is a major issue in performance debugging of parallel programs, and because existing solutions do not satisfactorily address this problem, there is a need for tools to help programmers efficiently find and eliminate causes of load imbalance in their code.

This section presents LIME, a framework that uses profiling, statistical analysis, and control flow graph analysis to automatically determine the nature of load imbalance problems and pinpoint the code where the problems are introduced. Unlike prior work that addresses load imbalance, LIME does not aim to automatically exploit or reduce load imbalance. Instead, it provides highly accurate information to

programmers about what is causing the imbalance and where in the code it is introduced, with the goal of minimizing trial-and-error diagnosis and the programming effort needed to alleviate the problem.

We built and evaluated our LIME framework on 15 parallel sections from SPLASH-2 [52] and PARSEC [5] benchmark suites, which are commonly used to evaluate performance of multi-processor and multi-core machines. Our results show that LIME is highly accurate in pinpointing load imbalance problems caused by cache misses and control flow differences among threads. We confirmed the accuracy of the tool's cache miss results by eliminating misses it reports and confirming that doing so dramatically reduces load imbalance. We confirmed the accuracy of the tool's control flow results by verifying that, when the tool reports control flow differences as the primary cause of load imbalance, the reported line of code is the actual location where the load imbalance is introduced and leads the programmer to the code where it can be repaired.

3.1 Overview of LIME

To help explain our framework and the problem it addresses, we use the code example in 28. This is an actual parallel section from SPLASH-2's *radix* benchmark [52]. The SPLASH-2 benchmark suite was extensively optimized over a decade ago by experts in both parallel programming and multi-processor hardware, and has been used to evaluate parallel performance of multi-processor and multi-core machines ever since.

```
534     BARRIER(...);
535     if (MyNum != (...)) {
540       while ((offset & 0x1) != 0) {  ...  }
549       while ((offset & 0x1) != 0) {  ...  }
557       for (i = 0; i < radix; i++) {  ...  }
560     } else {
562     }
566     while ((offset & 0x1) != 0) {  offset=...  }
575     for(i = 0; i < radix; i++) {  ...  }
578     while (offset != 0) {
579       if ((offset & 0x1) != 0) {
582         for (i = 0; i < radix; i++) {  ...  }
585       }
589     }
590     for (i = 1; i < radix; i++) {  ...  }
594     if ((MyNum == 0) || (stats)) {  ...  }
598     BARRIER(...);
```

Figure 28. Code for sample parallel section from *radix*. All non-control-flow statements are removed.

This parallel section begins and ends with barriers, where each thread waits for all others to arrive before proceeding further. Any load imbalance will result in threads arriving at the end barrier (line 598) at different times, forcing early-arriving threads to wait (i.e., be idle) until the longest-running thread arrives.

Within the parallel section, each thread uses its private *MyNum* and *offset* values to decide which part of the parallel computation it should perform. Depending of these values, the threads may have different execution times, either by executing different code (due to differences in control flow) or by taking different amounts of time to execute the same code (due to differences in how the executed code interacts with the hardware). In 28, there are several examples of possible control flow differences: if-then-else blocks at lines 535, 579, and 594 may cause only a subset of threads to execute the if-path (and for line 560, other threads to execute the else-path); loops at lines 540, 549, 557, 575, 578, 582, and 590 could all execute a different number of iterations for different threads. Nesting of loops and conditionals (e.g., line 582) can compound these differences. Additionally, threads may have differences in interacting with the system, such as branch predictor performance (some threads might have more predictable branch decision patterns than others) or cache performance (e.g., threads that access data already in the cache may have more cache hits). These differences are too numerous to point out even in our small example code because every branch, jump, load, store, etc. instruction in the compiled code may be, at least in theory, a potential source of these performance differences.

It may seem that a trained programmer can inspect the source code to identify potential causes of imbalance and repair them. However, this is a very labor-intensive and error-prone endeavor because of the sheer number of potential causes, and because of the complexity of understanding each cause and determining whether or not each is responsible for imbalance (and then repairing those that are).

An actual example of the threads' execution times for one dynamic instance of this parallel section is shown in 29. The shaded part of each bar represents the useful execution time of each thread, normalized to the overall execution time of the parallel section. The white part of each bar represents the thread's waiting time at the end of the parallel section (line 598).

Our discussion of possible causes of imbalance included two types of control flow causes—iteration counts of loops and decision counts of if-then-else blocks. These event counts are also shown in 29, with each event's counts normalized to the maximum count for that event among all threads. From visual inspection of the graph, the decision at line 535 appears to cause imbalance between the first thread and the others. The differences in "true" decisions at line 579 appears to account for the remaining imbalance (note how well the useful execution time tracks this factor for threads 2 through 8). The loops at lines 575 and 590 have identical iteration counts in all threads, and thus cannot be causing any imbalance—code inspection reveals the same insight, because the value of *radix* is constant and the same for all threads. The loops at lines 566 and 578 do produce different iteration counts in different threads but this does not seem to correspond to actual imbalance.

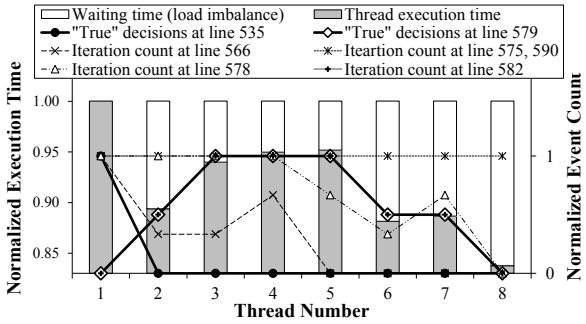


Figure 29. Execution time, imbalance, and thread behavior of key points in 28. Points that cause load imbalance are shown with thicker lines.

Finally, loop iteration counts at lines 540, 549, and 557 (not shown) have the same relationship with the imbalance that decision count at line 535 has, and loop iteration count at line 582 has the same behavior as the decision count for line 579.

Our LIME framework performs this kind of analysis automatically and quantitatively. For each thread in each parallel section, LIME measures execution time and various *event counts*. The event counts are dynamic decision counts for all static control flow decision points (control flow events), as well as dynamic counts for each static code location that causes machine-interaction events[3] (hardware events). Using this data, LIME's analysis framework determines how much imbalance exists, which control flow decisions and machine interaction events are related to the imbalance, and assigns scores that help programmers decide which cause of the imbalance to "attack" first.

Our initial implementation of LIME includes two different profiling environments. The first implementation uses a cycle-accurate hardware simulator called SESC [43] that can be relatively easily extended to collect any desired machine-interaction event count; however, since it performs detailed simulation of a computer system, it is very slow and can only be used for parallel sections that execute quickly (e.g., with carefully designed small input sets). To overcome the speed limit, we implemented LIME with Pin [36], which is fast but can only accurately collect data for analysis of control flow causes of imbalance. The simulator-based implementation was designed to let us experiment with collecting different events, and the Pin-based one to let us test LIME on larger input sets. For practical use, a purpose-built profiler could be employed to collect control flow events and key hardware performance counters more efficiently.

The analysis part of the framework processes profiling data from either profiling environment. It first clusters together events whose counts are highly correlated to each other. The purpose of this step is to group together events that seem to be related to the same potential cause of imbalance. For example, this step puts the decision count from line 535 and the iteration counts from lines 540, 549, and 557 in the same cluster because they are linearly related to each other (they have zero counts in all threads but one, so one of these event counts is equal to a constant times the execution count of another). Similarly, the iteration count from line 582 and the number of cache misses at lines 580–585 are in the same cluster as the "true" decision count from line 579 (this would not be true if threads had differing cache miss rates for lines 580–585).

Next, LIME finds the "leader" of each cluster. The purpose of this step is to identify the event that corresponds to "introducing" a potential cause of imbalance. In our two example clusters, the "true" decision counts from lines 535 and 579 are found to be the leaders of their respective clusters.

The next step in LIME uses multiple regression to find which cluster leaders are related to the imbalance in a statistically significant way, and to find the strength of that relationship. In our example, the "true" decision counts from lines 535 and 579 are the only cluster leaders to have a statistically significant relation to the load imbalance.

Finally, LIME ranks and reports the cluster leaders that are related to the imbalance. The report includes the score, the location in the code, and the corresponding cause of imbalance. In our example, LIME reports only two causes, both with relatively high scores: (1) threads take different paths at line 535, and (2) threads have different biases ("true" vs. "false" decision) for the if-then-else at line 579.

In the rest of this section we provide a detailed description of the analysis framework, followed by a description of the two profiling implementations we use, an experimental evaluation of LIME's accuracy with examples that provide more intuition about how LIME works and how its results can be used by the programmer to reduce load imbalance.

3.2 LIME Analysis Framework

LIME's analysis starts with data gathered by one of our profiling implementations (see 3.3). The profiling data consists of 1) per-thread control flow event counts for each edge in the static control flow graph and 2) per-thread hardware event counts for each static instruction that can cause such an event.

3.2.1. Causality Analysis for Hardware Events. Before entering the main analysis routine, LIME conducts preprocessing on collected hardware events in order to establish causal relationships between related events. For example, an L1 cache miss can occur only when a

[3] Among machine-interaction events, we only experimented with cache misses because we expected them to be the only machine-interaction event that plays a significant role in creating load imbalance. As will be shown in 3.4.4, this turned out to not always be true. However, LIME's analysis treats all hardware events in the same way and we expect it to readily extend to other events (as long as they can be counted efficiently).

memory access instruction is executed, an L2 cache miss can occur only when an L1 cache miss happens, etc. If the dynamic count for a particular hardware event differs among threads, this hierarchy among events allows us to split the "blame" for the difference between that event itself and the events that must precede it. For instance, when threads have different numbers of L1 misses at a particular static instruction, this may due to some threads executing that instruction more times (and thus having more L1 miss opportunities) or due to how the application interacts with the L1 cache. In the preprocessing step, LIME removes from subordinate hardware events (e.g. L1 misses) the contribution from their "superior" events (e.g. instruction's execution count) using the Gram-Schmidt process [19], leaving each event count only with the event's own contribution to variations among threads. This adjusted hardware event count is used instead of the naïve one throughout the LIME analysis.

3.2.2. Hierarchical Clustering. The second step in the LIME analysis is to cluster related events together. There are two commonly used clustering algorithms: hierarchical clustering and K-means clustering [27]. We use hierarchical clustering because it can automatically find an appropriate number of clusters for a given separation principle (clustering threshold) between clusters, whereas K-means yields a predetermined number of clusters. For the same reason, many projects on workload characterization [4, 29, 42] rely on hierarchical clustering to find benchmarks that have similar behavior.

Hierarchical clustering is performed in steps. Each step merges the two clusters that are "closest" according to a distance metric. Clustering ends when the distance between the two closest clusters is larger than a preset threshold.

In LIME, each event (control flow event or hardware event) is initially a cluster. We then compute a proximity matrix in which an element (i, j) represents the distance between 'cluster i' and 'cluster j'. At each step, we merge the two closest clusters and update the proximity matrix accordingly. The distance metric LIME uses is Pearson's correlation between the event's per-thread counts, because it effectively captures similarity in how the event count behaves in different threads. For linkage criteria, we used average linkage (UPGMA [40]), but other methods (single/complete-linkage) produced similar results.

LIME stops clustering when the largest correlation is $<$0.9. This threshold value provides the best results in most parallel sections we tested. The exceptions are *fmm* and *fluidanimate*, where we used threshold values of 0.8 and 0.6, respectively. A poorly chosen threshold affects the usefulness of the report—too-high of a threshold prevents merging of correlated event clusters, while too-low of a threshold results in clusters that contain unrelated events.

Clustering provides several benefits for further analysis:

- It results in a major reduction in the number of subjects for further analysis.
- It gathers highly co-linear events into one cluster, which improves accuracy of regression[4] in 3.2.5.
- It helps identify significant decision points in the program structure (e.g., branches where control flow differs between threads), as we explain in 3.2.4.

3.2.3. Classification of Clusters. Clusters are classified into two types: those that contain control flow events (control-flow clusters), and those with only hardware events (hardware event clusters).

For hardware event clusters, the absence of highly correlated control flow events indicates that different threads suffer the hardware events differently for the same code. Therefore, if any load imbalance is eventually attributed to the cluster, all the hardware events in the cluster are reported as contributing to that portion of the imbalance.

3.2.4. Finding Cluster Leaders. Within a control-flow cluster, control flow events are typically interdependent. To improve the usefulness of reported results, for each cluster, LIME discovers a *leader node*—a control flow instruction that steers program execution into the cluster. Intuitively, if the cluster is related to the imbalance, the leader node represents the code point where this imbalance is introduced.

Leader nodes are important because they are the decision points that change thread execution characteristics. They are the points in the program of most interest to the programmer: by inspecting the code that affects the leader node's decision, the programmer can typically find the high-level reason for the imbalance.

To formally define a leader node, assume a control flow graph of a program has vertices V and edges E. The leader node of a cluster C is a vertex v in the control flow graph such that all incoming edges to v *except backedges* have source vertices outside the cluster, i.e., $v \in C$ such that $\forall (s, v) \in E, s \notin C$.

Examples of typical clusters and their leader nodes are shown in 30. The leftmost example shows a cluster whose leader is an *if*-statement. The middle example shows why a backedge restriction is needed in the leader definition—it allows a loop cluster to have a leader (the loop entry point). The rightmost example shows a cluster with two leaders (A and G)—this typically occurs when the same control flow decision is made in more than one code point.

After finding leader nodes for each cluster, each leader node is assigned a score according to its significance in creating load imbalance. First, LIME computes, for each edge, the Pearson's correlation coefficient between that edge and the execution time. The score of a leader node is the *difference* in this correlation between the node's incoming and outgoing edges (again, ignoring backedges). For example, if n threads have execution times $T = (t_1, t_2, \cdots, t_n)$ and a leader node v has incoming edge counts $ie_{v_1}, ie_{v_2}, \cdots, ie_{v_i}$, and outgoing edge counts $oe_{v_1}, oe_{v_2}, \cdots, oe_{v_j}$, the score s_v of the leader node v is

$$s_v = \max_x \big(corr\left(oe_{v_x}, T\right)\big) - \max_y \big(corr\left(ie_{v_y}, T\right)\big) \tag{1}$$

[4] Statistical regression works poorly with collinear vectors.

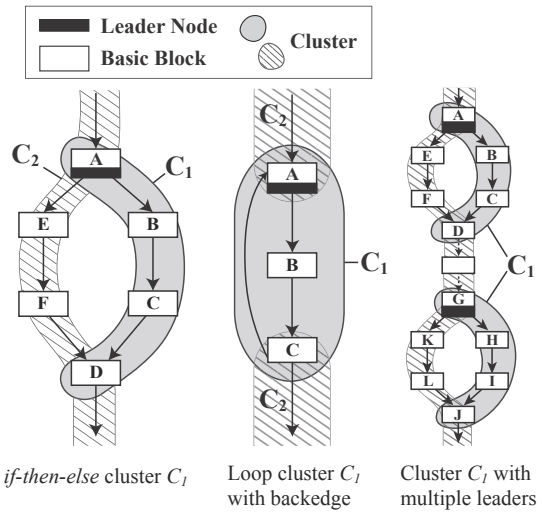


Figure 30. Example of clusters and leader nodes

where $corr(a, b)$ is the Pearson's correlation coefficient between vectors a and b.

Intuitively, score s_v measures the amount of correlation the leader node incurs in regard to the overall imbalance. A high score means that the node converts events that are unrelated to load imbalance into events that are highly correlated to the imbalance.

3.2.5. Multiple Regression. Multiple regression analysis [16] is a statistical technique that estimates the linear relationships between a dependent variable and one or more independent variables. In LIME, the dependent variable is the vector of per-thread execution times, and the independent variables are the clusters. If a cluster appears to be a good predictor of execution time, we can infer with high confidence that the cluster is responsible for the differences in execution time between threads.

For the regression analysis, we need to combine event counts of all events in the cluster so that the cluster behaves like a single event. For this, LIME uses averaged Z-scores [32] of events in a cluster. A Z-score standardizes all events to have the same average and same variation, so that all events carry equal weight in determining the cluster's overall "event count", regardless of the actual absolute event counts for each event. For each event with a mean per-thread dynamic count μ and a standard deviation σ, for each per-thread dynamic count x, the Z-score is $z = \frac{x-\mu}{\sigma}$. After this step, regression proceeds by treating each cluster as a single "event" whose per-thread "event counts" are the cluster's per-thread Z-scores.

During regression analysis, LIME excludes clusters that are statistically redundant or insignificant in building a regression model to explain the execution time. We use the forward selection method [16] to choose which clusters to include in the model. The method selects clusters based on their unique contribution to the variance in the dependent variable (execution time). LIME iteratively adds the selected clusters to its regression model until none of the remaining clusters is statistically significant (determined by F-test [35]).

Multiple regression analysis computes a standardized coefficient, β_C, for each cluster C, which represents how sensitive execution time is to that cluster. That is, the regression computes the values of β_{C_i} to best fit $T \approx \Sigma\beta_{C_i} \cdot C_i$, where T is the vector of per-thread execution times and C_i is the vector of averaged Z-scores for cluster i across all events in the cluster. LIME uses the β_C values as a measure of a cluster's importance for load imbalance as follows.

For each control-flow leader node and hardware event, LIME computes a *final score*—an estimate of how responsible that node/event is for load imbalance. The score is based on regression results and, for control-flow clusters, on leader node importance scores. For a leader node v_i in control-flow C_j, the final score fs_{v_i} is

$$fs_{v_i} = \beta_{C_j} \cdot s_{v_i} \tag{2}$$

where s_{v_i} is the score (from 1) for leader node v_i, and β_{C_j} is the standardized coefficient (from regression) for cluster C_j. For a hardware event (e.g., cache miss) h_i in a hardware-event cluster C_j, the final score fs_{h_i} is equal to β_{C_j}.

31 shows an example of computing final scores for two control-flow clusters. Cluster C_1 has one leader node A, while cluster C_2 has two leader nodes A and G. Note that node A is a leader in both clusters.

In this example, node G from cluster C_2 has the highest final score (i.e., a leader node score of 0.82 and a β_{C_2} of 0.91 combine to give a score of 0.7462). Therefore, we conclude that cluster C_2 is important in explaining the imbalance, and that node G is the prime suspect for introducing the imbalance.

3.2.6. Reporting to the Programmer. In general, a program runs a static parallel section multiple times during its dynamic execution; LIME analysis operates on each dynamic instance independently because imbalance and other characteristics may vary across instances. Since programmers prefer feedback on static code, LIME then combines results from all dynamic instances of each parallel section using a weighted average, with load imbalance of a dynamic instance serving as its weight. For each leader node in a control-flow cluster, and for

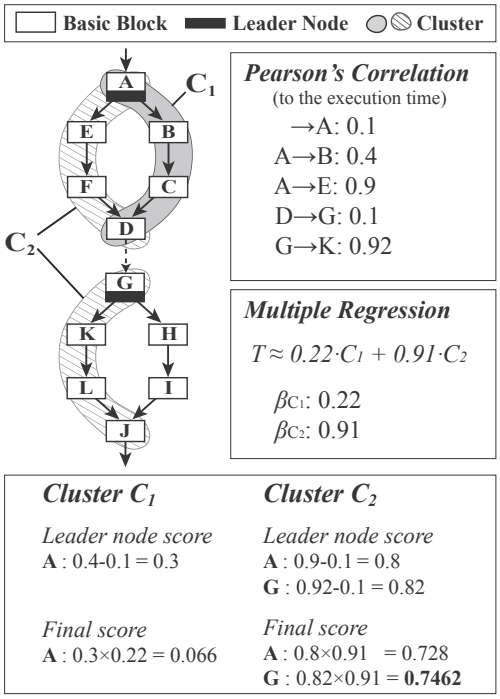


Figure 31. Example of final score computation

Benchmark	Suite	Parallel Section	Type	Input Size	Imbalance			
					8 Core	16 Core	32 Core	64 Core
LU	SPLASH-2	lu.c:604	Barrier	1024×1024 matrix, 64×64 block	16.1%	35.5%	92.3%	92.3%
volrend	SPLASH-2	main.C:267	Barrier	head	15.3%	25.8%	38.8%	46.9%
fmm	SPLASH-2	fmm.C:283	Barrier	16,384 particles	10.6%	13.7%	22.9%	26.2%
barnes	SPLASH-2	code.C:715	Barrier	16,384 particles	10.2%	11.6%	13.5%	16.7%
canneal	PARSEC	annealer_thread.cpp:88	Barrier	10000 swaps/step, 32 steps	3.8%	6.9%	12.7%	15.2%
fluidanimate	PARSEC	pthreads.cpp:793	Barrier	35K particles	8.8%	15.3%	19.5%	13.3%
blackscholes	PARSEC	blackscholes.c:374	T. Join	4,096	0.2%	1.1%	7.0%	12.7%
water-sp	SPLASH-2	interf.C:205	Barrier	512 molecules	1.7%	6.6%	10.7%	12.0%
radix (small)	SPLASH-2	radix.C:497	Barrier	2,097,152 integers (small input)	0.2%	0.5%	3.4%	5.7%
swaptions	PARSEC	HJM_Securities.cpp:271	T. Join	16 swaptions, 5000 simulations	0.0%	1.2%	1.9%	3.3%
radix	SPLASH-2	radix.C:497	Barrier	4,194,304 integers	0.0%	0.1%	0.2%	3.3%
ocean	SPLASH-2	slave2.C:812	Barrier	514×514 grid	1.3%	1.3%	1.5%	1.5%
fft	SPLASH-2	fft.C:623	Barrier	4,194,304 data points	0.1%	0.7%	1.5%	1.5%
streamcluster	PARSEC	streamcluster.cpp:706	Barrier	4,096 points	1.1%	1.4%	1.4%	1.1%
radiosity	SPLASH-2	rad_main.C:810	Barrier	room	0.1%	0.3%	0.4%	0.6%
Average					4.6%	8.1%	15.2%	16.8%

Table 3. Description of applications and parallel sections used in our experiments. Parallel section denotes the location of the *pthread_barrier_wait* or *pthread_join* call that delimits the parallel section. Imbalance is the average percentage of idle time threads spend in the parallel section.

each hardware event (e.g., cache misses at static code location Y) in a hardware-only cluster, LIME presents to the programmer its static code location, type of event, and weighted score.

3.3 Profiling Implementation

The LIME framework consists of two parts: (1) collection of profiling data, and (2) analysis of the data. Our contribution is primarily in the LIME analyzer, and our goal is to demonstrate the utility of our novel analysis techniques. We built the analyzer in C++ using the armadillo linear algebra library [44].

There is a large body of prior work on profilers. Our framework can make use of any data profiler that can collect the required control flow and hardware event counts. For our prototype tool, we implemented two different versions of the profiler.

Our first profiler is built on Pin [36], a binary instrumentation tool. With this profiler, we can collect control flow events, but no hardware events. We instrument synchronization points such as barriers to identify parallel sections in our program. We also instrument all branch instructions, and gather edge (control flow event) counts using a hashmap structure. When branch instruction b is executed and the previous branch was p, we increment the entry for edge (p, b). Since dynamic instrumentation severely distorts execution time, we use instruction

Benchmark	8 Cores				16 Cores				32 Cores				64 Cores			
	Ctrl flow		Cache miss		Ctrl flow		Cache miss		Ctrl flow		Cache miss		Ctrl flow		Cache miss	
	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score
LU	2 (8)	**0.97**	0 (1)	0.01	1 (4)	**0.94**	0 (3)	0.05	1 (1)	**0.88**	3 (9)	0.14	1 (1)	**0.72**	3 (9)	0.29
volrend	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–
fmm	1 (2)	**0.83**	4 (20)	0.12	1 (3)	**0.51**	0 (11)	0.06	1 (5)	**0.50**	0 (25)	0.06	1 (6)	**0.36**	0 (21)	0.04
barnes	1 (1)	**1.00**	0 (0)	–	1 (1)	**0.86**	0 (3)	0.05	1 (5)	**0.89**	0 (3)	0.01	3 (3)	**0.64**	0 (17)	0.07
canneal	1 (4)	**0.24**	0 (18)	0.04	1 (4)	**0.45**	0 (21)	0.05	1 (2)	**0.39**	1 (12)	0.21	1 (2)	**0.32**	1 (16)	0.12
fluidani.	1 (4)	**0.69**	0 (3)	0.03	1 (3)	**0.20**	0 (3)	0.03	1 (1)	**0.39**	1 (15)	0.15	1 (1)	**0.19**	0 (12)	0.02
water-sp	1 (1)	**0.93**	0 (0)	–	1 (1)	**0.92**	0 (0)	–	1 (1)	**1.06**	0 (0)	–	1 (1)	**0.86**	0 (0)	–
streamcls.	3 (3)	**0.50**	0 (0)	–	3 (3)	**0.50**	0 (0)	–	3 (3)	**0.50**	0 (0)	–	3 (3)	**0.50**	0 (0)	–
radiosity	2 (2)	**1.00**	0 (0)	–	2 (2)	**0.99**	0 (1)	0.03	2 (2)	**0.95**	0 (1)	0.03	2 (2)	**0.78**	0 (1)	0.06
blackschls.	0 (0)	–	4 (4)	**0.49**	0 (0)	–	0 (0)	–	0 (0)	–	8 (19)	**0.92**	0 (0)	–	9 (14)	**1.00**
radix (small)	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**
radix	0 (0)	–	1 (1)	**0.99**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (2)	**0.99**	0 (0)	–	1 (1)	**1.01**
ocean	0 (0)	–	1 (1)	**0.97**	0 (0)	–	1 (3)	**0.98**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (3)	**1.06**
swaptions	0 (0)	–	0 (0)	–	0 (0)	–	0 (1)	0.07	0 (0)	–	2 (8)	**0.41**	0 (0)	–	0 (5)	0.08
fft	0 (0)	–	2 (2)	**1.03**	0 (0)	–	3 (3)	**0.61**	0 (0)	–	4 (4)	**0.81**	0 (1)	0.04	2 (2)	**0.37**

Table 4. Results of LIME analysis. The Rpt columns show the number of reported events with score greater than 0.1. The number of all reported events is listed in parentheses. The Score columns give the reported score for the highest-scored event of each type.

count as an approximation to the execution time. When a dynamic parallel section ends (i.e., a thread enters a barrier), recorded data is exported to output files for later processing.

Our second profiler is built on SESC [43], a cycle-accurate computer system simulator. This simulator functionally emulates an application and uses the instruction stream to drive a detailed model of the caches, memory, and processor cores, including the key microarchitectural structures (e.g., the instruction queue and reorder buffer, which allow for out-of-order execution). This simulator is routinely used by computer architects to evaluate architectural and microarchitectural proposals. We use the simulator to collect cache miss event counts and control flow edge counts simultaneously. Event recording and data exporting is very similar to our Pin-based tool (e.g., control flow edges are counted with a hashmap). The primary difference is that our SESC-based tool can accurately measure execution cycles and hardware event counts, but is much slower than the Pin-based version.

3.4 Experiments and Results

In this section, we present our experimental setup and the output of our LIME analysis framework, and then verify the output to show that LIME reports imbalance-related performance bugs accurately.

3.4.1. Experimental Setup. We tested LIME using data gathered from 15 parallel sections in multithreaded applications from SPLASH-2 [52] and PARSEC [5] benchmark suites. For runs with the SESC-based profiler, we simulate a typical multi-core general purpose processor with cores at 1GHz, 16KB each of data and instruction cache per core, and 4MB of shared cache. Since the accuracy of LIME depends on the number of threads (i.e., sample points), we run our analysis on a varying number of cores (8, 16, 32, and 64) to verify that our scheme can find performance bugs accurately across a wide range of available cores. In all configurations, we run one application thread on each core.

3 summarizes the location and type of each parallel section. We sort the sections in order of decreasing imbalance on 64 cores. The tested parallel sections cover a wide range of imbalance, from almost no imbalance (radix for 8 cores) to over 90% imbalance (LU for 64 cores).

3.4.2. Simulator-Based Profiling. 4 summarizes the results of LIME using the SESC-based profiler. The "Rpt" columns show the number of "important" events (those with a report score greater than 0.1) for both control flow events and cache miss events. The "Score" column shows the highest score reported among control flow and among cache miss events. The larger of the two highest scores is shown in bold font, to emphasize the event type that is more important according to LIME.

For all but two of the parallel sections, LIME is able to draw a consistent and clear conclusion on the most important event type. For the top nine sections, control flow events cause the load imbalance, while for the next four, cache miss events cause the imbalance. For the bottom two parallel sections, LIME fails to find a consistent cause of imbalance; this is because, in our current profiling implementation, we only collected control flow and cache miss event counts, which do not cause the imbalance for these benchmarks. We note that this is not a limitation of the LIME analysis framework, but rather a limitation of the even count profiling implementation. 3.4.4 explains the true cause of imbalance for these two parallel sections and how it affects LIME.

LIME consistently reports a small number of important events; this is important because it means the programmer should be able to inspect the spots in the source code that LIME reports. On average, for benchmarks with imbalance from control flow, it reports 1.3 important control flow instructions per benchmark. For benchmarks with imbalance from cache misses, on average LIME reports 2.1 important code points per benchmark.

3.4.3. Pin-Based Profiling. The limitations of the Pin-based implementation are that it introduces significant distortion in thread execution times and that it cannot accurately capture all the hardware interaction events that might be causing load imbalance. However, it can gather control flow edge information at speeds that allow "real-world" problem sizes.

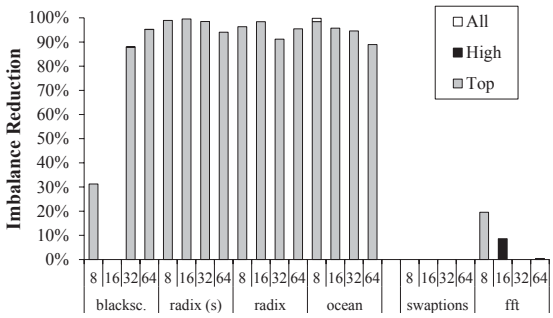


Figure 32. Imbalance reduction of each parallel section when reported cache misses are eliminated.

To circumvent execution time distortion, in Pin-based profiling the instruction count is used as a proxy for execution time. This eliminates the possibility of identifying hardware-interaction events as causes of imbalance, but this profiler does not collect those events anyway. Further, the instruction count may not accurately represent the execution time. We validate that this profiler is useful in quickly identifying control flow sources of load imbalance. If hardware events trigger additional imbalance, a more expensive simulator-based approach can be used.

Overall, the control flow results of Pin-based profiling are very similar to those from the simulator. For brevity, 5 shows LIME results for only five parallel sections for "real" inputs (too large for simulation), including one from a benchmark (PLSA from bioParallel benchmark [28]) that is infeasible to run in simulation. Also shown are simulation-size inputs for three benchmarks for comparison, with scores from simulator-based profiling shown in parentheses. The profiling is done on an Intel Quad Core Xeon server using 8 threads (four cores, each with support for two threads).

Benchmark	Input Size	Slowdown	Ctrl Flow	
			Rpt	Score
LU	1K×1K matrix	21.1×	3 (10)	**0.97** (0.97)
	16K×16K matrix	23.9×	3 (6)	**0.93**
barnes	16,384 particles	84.1×	1 (6)	**0.96** (1.00)
	1,048,576 particles	88.7×	2 (5)	**0.97**
streamcluster	4,096 points	23.0×	2 (2)	**0.75** (0.50)
	1 million points	4.2×	2 (2)	**0.75**
radiosity	largeroom	62.6×	2 (3)	**1.00**
PLSA	100K sequence	127.0×	1 (13)	**1.36**

Table 5. Results with Pin-based profiling.

3.4.4. Verifying Reported Cache Misses. While we have already shown that our framework consistently identifies a programmer-manageable number of causes of load imbalance, we now verify that our framework (with the SESC profiler) *correctly* identifies the causes of load imbalance, starting with cache misses. To verify that reported cache miss events are indeed causing load imbalance, one possible approach would be to try to reorganize the data structures and algorithms in each application, re-evaluate parallel performance, and check if load imbalance has been reduced. However, we lack domain expertise and resources to make such extensive changes in so many applications. Further, the results would highly depend on how well we understood each application, data structure, and algorithm.

Instead, we use cycle-accurate simulation to artificially eliminate the reported cache misses, while leaving all other aspects of the execution intact. To "erase" misses from reported memory access instructions, we override the simulated cache behavior for that instruction to make each dynamic instance of that instruction into a cache hit. If the reported cache misses are indeed the source of the imbalance problem, this modified execution should have a dramatically reduced load imbalance.

32 shows the results of this simulation for applications in which LIME reports cache misses as the main cause of load imbalance. The shaded portion of each bar represents the imbalance removed when only "erasing" the cache misses caused by the highest-scoring instructions for each application. The black portion represents additional imbalance removed when also "erasing" misses from all memory access instructions reported with final scores above 0.1. The white portion is the additional imbalance removed when erasing all cache misses reported by LIME as statistically significant causes of load imbalance.

For the first four applications in 32, load imbalance is reduced dramatically when LIME-reported cache misses are "erased," except for configurations that do not have significant load imbalance to begin with (8 and 16 core configurations for blackscholes). It is important to note that, in all these simulations, we end up "erasing" misses from only 39.6% of all dynamic memory accesses, and the observed imbalance reduction is *not* caused simply by a dramatic reduction in execution time—in fact, the speedup in these runs mostly comes from reducing imbalance (making the slowest threads finish faster), with little performance benefit for the fastest threads.

Recall from 3.4.2 that LIME did not find a consistent cause of imbalance for the last two applications because neither control flow nor cache misses are the true cause. Further investigation reveals that imbalance is actually caused by cores having different success in getting access to the L1–L2 on-chip (back-side) bus when servicing cache misses—in these applications, this bus has high utilization and the bus arbitrator seems to be favoring some cores at the expense of others. We confirm this by simulating a higher-bandwidth bus (without "erasing"

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x4018b4	0.880	lu.C:668 (lu)

Reported source lines in *lu.C*

```
668     if (BlockOwner(I, J) == MyNum) {   /* parcel out blocks */
669         B = a[K+J*nblocks];
670         C = a[I+J*nblocks];
671         bmod(A, B, C, strI, strJ, strK, strI, strK, strI);
672     }
```

```
556     long BlockOwner(long I, long J)
557     {
558         return((I + J) % P);   // P: number of threads
559     }
```

Figure 33. LIME report for LU.

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x4068ec	0.999	render.C:38 (Render)

Reported source lines in *render.C*

```
31    Render(int my_node)         /* assumes direction is +Z */
32    {
33        if (my_node == ROOT) {
34            Observer_Transform_Light_Vector();
35            Compute_Observer_Transformed_Highlight_Vector();
36        }
37        Ray_Trace(my_node);
38    }
```

main.C

```
298     Render(my_node);
300     if (my_node == ROOT) {
307         WriteGrayscaleTIFF(outfile, image_len[X],  ... );
310         WriteGrayscaleTIFF(filename, image_len[X],  ... );
312     }
```

Figure 34. LIME report for *volrend*.

any misses) and observing a 95% imbalance reduction. However, unlike cache miss (and many other) events that can easily be attributed to particular instructions, such attribution for bus contention events is an open problem that we have not addressed in this project.

3.4.5. Verifying Reported Control Flow Causes. To verify that LIME correctly reports control flow events that cause imbalance, we use a different methodology than for cache misses—control flow events cannot be "erased" without affecting many other aspects (including correctness) of program execution. Thus, in all parallel sections where LIME reported control flow code locations as a significant cause of imbalance (4), we manually confirm that the location and nature of the problem was correct. For lack of space, we only describe this analysis for three examples, selected to both illustrate different types of imbalance causes and to only require brief code fragments for explanation.

Among the parallel sections used in this evaluation, *LU* has the most imbalance—the last-arriving thread takes over ten times longer than the first-arriving thread.

For the 32-core configuration, our framework reports only one static instruction, shown in 33, as statistically significant cause of imbalance (at line 668). In the other three configurations, LIME also reports line 668 as the top-scoring (by a large margin) cause of imbalance.

The corresponding code point for the top-scoring instruction is also shown in 33. This is an *if*-statement that parcels out blocks to threads using function *BlockOwner*, which returns the summed block coordinates modulo number of threads, P. Intuitively, this method of assigning blocks to threads should produce a balanced load. However, values of *I* and *J* vary in a range determined by the program's inputs—they may be small and/or not a multiple of P, causing uneven distribution of blocks to threads. For example, when *I* and *J* take values 1 through 15, the 225 blocks are distributed among 32 threads and ideally each thread should get about 7 blocks. The actual assignment using *(I+J)%P* turns out to assign only one block to threads 2 and 30, 2 blocks to threads 3 and 29, etc., giving 15 blocks to thread 16.

When line 558 in the *BlockOwner* function is changed to "return $(J\%Ncols) + (I\%Nrows) \times Ncols$;", where Ncols and Nrows are 8 and 4, respectively, for a 32-core configuration, the assignment of blocks to threads becomes more balanced and results in eliminating 61% of the original load imbalance. This confirms that imbalance was indeed introduced at the code point reported by LIME.

Our second verification example is from *volrend*, which has up to 46.9% imbalance on 64 cores. The LIME report for 32 cores, summarized in 34, suggests with high confidence that the return point of the function *Render* is the source of imbalance. At the first glance, this looks like a false report, but a closer inspection reveals that the problem is the mapping of the instruction to the line of source code because the compiler obfuscated the situation via inlining and other optimizations. When we examine the code in a disassembler, the reported control flow instruction is actually the *if*-statement at line 300 (immediately after the callsite for *Render*). This *if*-statement assigns extra work to the main thread. When compiled without optimizations, LIME reports the *if*-statement in line 300 correctly.

Our third verification example is from *Barnes*, with 13.5% imbalance on the 32-core configuration. The LIME report for 32 cores, summarized in 35, says the control flow edge from line 116 in *grav.C* to line 112 accounts for the imbalance. This edge corresponds to the

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x405470	0.893	grav.C:116 -> grav.C:112
2	0x40548c	0.030	grav.C:113 (walksub)
3	0x4055cc	0.024	grav.C:136 -> grav.C:114

Reported source lines in *grav.C*

```
105    void walksub(nodeptr n, real dsq, long ProcessId)
106    {
107       nodeptr* nn;
                :
112       if (subdivp(n, dsq, ProcessId)) {  // First branch in walksub
113          if (Type(n) == CELL) {
114             for (nn = Subp(n); nn < Subp(n) + NSUB; nn++) {
115                if (*nn != NULL) {
116                   walksub(*nn, dsq / 4.0, ProcessId);
117                }
118             }
```

Figure 35. LIME report for *barnes*.

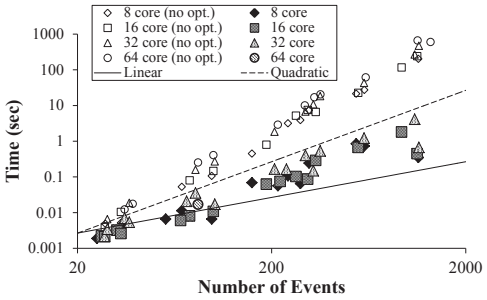


Figure 36. Clustering time vs. number of events. Each point represents clustering time of one parallel section. Note the logarithmic scale markings on each of the axes.

recursive function call to *walksub*—*Barnes* implements the Barnes-Hut approach for the N-body problem, and *walksub* recursively traverses the primary data structure, a tree. Since LIME reports the tree traversal is imbalanced, this suggests that the tree itself is imbalanced. Further investigation shows that 86.5% of the total imbalance comes from the first dynamic instance of the parallel section, because the tree is skewed at the beginning of the run. The first Barnes-Hut iteration rebalances the tree and the load imbalance decreases.

3.4.6. Scalability of LIME. While LIME ran fast enough for this study, two possible concerns are: (1) what happens to analysis time as the core count increases, and (2) what happens to analysis time as event count (i.e., program size) increases?

A single-threaded implementation of LIME analysis took an average of 1.4, 1.8, 3.2, and 5.0 seconds to analyze the parallel sections for 8, 16, 32, and 64 cores, respectively, on a 2.67GHz Intel Quad Core Xeon processor with 12GB memory. This time includes clustering, leader selection, and regression. Since the analysis time grows in sub-linear proportion to the core count, we expect that a parallel implementation of LIME analysis will have a favorable scaling trend as the number of cores (for both application execution and analysis) increases.

The most time consuming part of LIME is the clustering part of the analysis—it accounts for over 76% of the average analysis time when no optimization is applied. A naive implementation of our hierarchical clustering method has asymptotic complexity of $O(n^3)$ where n is the number of event counts—to create a near-constant number of clusters, LIME does $O(n)$ merges, and for each merge it recomputes all $O(n^2)$ entries in the new cluster proximity matrix. This would be a major problem for applying LIME to real applications. To accelerate the clustering algorithm, we can cache the proximity matrix and perform bulk clustering. When merging two clusters, only the proximity values involving those two clusters become useless. Therefore, we re-use (cache) the proximities and compute only $O(n)$ new values. The "bulk clustering" optimization merges multiple clusters per step—as we compute the proximity matrix, we track clusters that are very close to each other and merge all of them at once. In this study, we use a proximity threshold of 0.99 for bulk clustering. Bulk clustering reduces the number of merges, but runs the risk of producing less precise clustering. We did not experience any imprecise clustering in our experiments.

36 compares the speed of the clustering implementation with and without our acceleration technique using a log–log plot. While our optimized implementation is still $O(n^2)$ (bulk clustering does not reduce the asymptotic complexity), in practice it shows near-linear scaling with n. This demonstrates that LIME is scalable with program size. Further optimizations are certainly possible but we leave them for future experimentation and fine-tuning.

3.5 Related Work

Performance debugging has long been studied in distributed systems. Much work focuses on finding the causal trace, or the trace with the longest path through a distributed system; this is analogous to the slowest thread in our study. The causal trace naturally tells programmers where to focus optimization efforts. LIME shares the same goal as this work, but works in a different domain. Among the related literature, the performance debugging method proposed by Aguilera et al. [1] draws our attention—they use a black box approach that finds the causal

trace without any knowledge of the system, and a signal processing technique called convolution to infer causal relationships. LIME collects profile data without a programmer's intervention, and uses clustering and regression analysis to infer causal relations between events and load imbalance.

A number of tools exist to detect and measure parallel overheads and inefficiency (i.e., idleness). One recent example is from Tallent et al. [48]. Their goal is to pinpoint where parallel bottlenecks occur, and classify bottlenecks as overhead or idleness. Our work is largely orthogonal to this—once a programmer knows that a problem exists, they can use our framework to help find the cause for the bottlenecks they have detected. Tallent et al. also have work on analyzing lock contention [49]. This has a lot in common with our work since they try to detect the cause of lock contention and identify the lockholder to blame. Our work focuses on barriers rather than locks and provides more direct information about where in the code the problem arises.

There is also significant work on trying to optimize performance and energy in the presence of load imbalance. Thrifty Barrier [34] predicts how much slack (i.e., idle time) each thread will have using a history-based predictor, and saves power by putting non-critical threads into a sleep state. Meeting Points [8] uses a different predictor that counts thread deviation at checkpoints called meeting points. It delays non-critical threads using dynamic voltage and frequency scaling to save power. It also attempts to accelerate the critical thread by prioritizing it. The Thread Criticality Predictor [3] similarly predicts thread criticality based on adjusted cache miss rates, and prioritizes threads based on criticality. While automatically detecting and prioritizing critical threads requires no programmer effort, it can only reduce load imbalance by a limited amount. Instead, we help programmers find and permanently fix imbalance problems in applications regardless of how severe the problems are; this can (and usually does) have a dramatic performance impact.

4 Lock Contention Debugging

Lock contention can be reduced by either reducing the amount of work in each critical section, increasing the time spent outside critical sections, or by using finer-grain locking. Among these, the easiest approach for the programmer is finer-grain locking - the other two approaches usually require more significant code changes that take time and may introduce correctness issues. Our work focuses on two approaches to finer-grain locking: 1) identifying opportunities for using different locks in different static critical sections (this usually requires very little effort from the programmer, usually just a change in the lock variable in each code point where such lock splitting is done), and 2) identifying opportunities for using per-object locks where more coarse-grained locks were used.

4.1 Splitting Static Locks

Lock contention happens when multiple critical sections that share the same lock variable compete for an ownership of the lock. As shown in the left side of the Figure 37, contention defers an execution of a waiting critical section after the one of currently holding the lock, resulting elongated total execution time. With current trend of increasing number of cores, it is likely that the lock contention grows and impacts overall performance of application considerably. To alleviate the lock contention, we propose a systematic and automated way to split multiple critical sections that are guarded by the same lock variable. After the split, each multiple different lock variables would be exploited to protect different critical sections (right side in Figure 37). As a result, threads can execute multiple critical sections without any stall time and total execution time can be reduced.

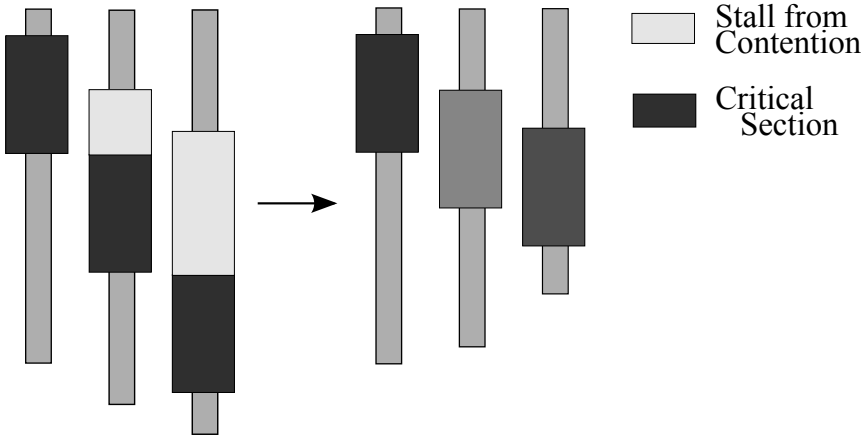


Figure 37. Lock Splitting Example.

One current tool only allows lock splitting when critical sections that will be guarded by different lock variables have no overlap in which memory locations they accesses. In general, this condition can be relaxed and we can leave it up to the programmer to ensure that multiple memory accesses to the same shared location are protected by the same lock variable. However, this process is cumbersome and error-prone so we decided to first work on recommending the safest and least error-prone approach.

Our tool starts with a profiling run of the target application, using a single pseudo-lock to guard all critical sections. The profiling run extracts which memory addresses are accessed during dynamic instances of each static critical sections and how they interact. Our framework requires those two types of information for different reasons. The memory access profile helps us preserve correctness by using the same lock for all critical sections that share any memory locations. Interaction allows us to reduce programmer effort by only splitting locks when such a split can be expected to provide significant performance gain. Although our framework suggests the best possible combination of locks that mitigates the contention, the programmer still has to decide whether the recommended lock separation is correct - our framework is based on dynamic analysis and can recommend incorrect lock separation if the profiling run had no overlap between some critical sections that may overlap when using different inputs. This is another reason for suggesting only those lock splits that lead to significant performance improvement - the programmer has to verify each split to ensure correctness.

After the profiling is done, our tool calculates two scores for each group of static critical section: "splittable" and "profitable". Two static critical sections are "splittable" if they share no common memory accesses during any of their dynamic instances. To facilitate understanding of the process, we represent each static critical section as a node in a graph, and the memory sharing as an edge between nodes if they have any. The "splittable" criterion is satisfied when two nodes are not connected by any edges.

The "profitable" criterion determines when it is beneficial for two critical sections to use two different lock variables. If dynamic instances of the two static critical sections tend not to execute at the same time, i.e. if there is almost no contention between them, the performance gain from using different lock variables in these critical sections will not be significant, so they are not "profitable" for lock splitting. Thus our "interaction" profiling measures how often a dynamic instance of one static critical section is waiting while the lock is held by that dynamic instance of the other critical section. This also results in a graph, but this time the edges represent the waiting-time interaction. Specifically, when a critical section waits while another holds the lock, we add that time to the weight of the edge between the two nodes. More precisely, the interaction is measured as follows.

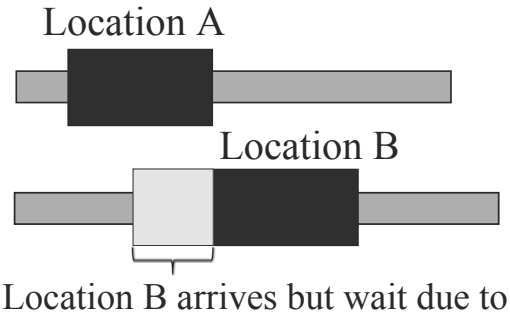


Figure 38. Calculation of interaction between two critical sections.

During execution, when a dynamic instance of a static critical section from code location B arrives but has to wait another critical section from location A because they share a same lock variable, we blame the stall time of B to A (Figure 38). Then, we assign the cumulative time that a static location B waits on location A as a weight of the edge going from node A to node B in the graph. The intuition behind the edge weights and graph generation is that, if those two critical sections are split to use different lock variables, the stall time represented by the edge between them would go away. More precisely, a split can be represented by a boundary that separates nodes protected by different locks, and the profitability (wait time reduction) of each cut can be approximated as the sum of weights of edges that are cut by that boundary. Given this, our goal of generating as few locks as possible while providing good performance can be transformed into splitting the graph nodes into as few groups as possible, while ensuring that no "memory sharing" edges cross between groups, and while maximizing the weights of "interaction" edges that cross between groups. Figure 39 gives an example of the graph generated for the raytrace application from the SPLASH-2 benchmark suite. In the graph, sharing edges are shown in red and interaction edges are shown in black (along with weights). Blue boxes denote groups that can use different locks (no sharing). Here, merging lock group 4 into lock group 2 won't affect performance much because the weight of the edge from B_409838 to B_40aaa8 is very low. Our simulation also shows that the performance difference is less than 1% when group 2 and group 4 are separated.

4.2 Generating Per-object Locks

To decrease lock contention even further, we can assigning locks to individual instances of objects accessed by the same static critical section. This allows different instances of a static critical section to use different locks, and thus eliminates contention created between these dynamic instances when they are used in different threads.

However, working with a large number of mutex locks has a cost. First, it is a burden for the programmer to manage the lifetime and use of each lock instance. Second, more lock variables use more memory and put more pressure on the caches. This can lead to performance degradation that negates the performance benefit from the effort put into introducing these lock instances. Finally, if a single dynamic instance of a critical section accesses data that is now protected by different locks, all of those locks must be held in that critical section to ensure correctness, which leads to additional overhead and the introduces the possibility of a deadlock.

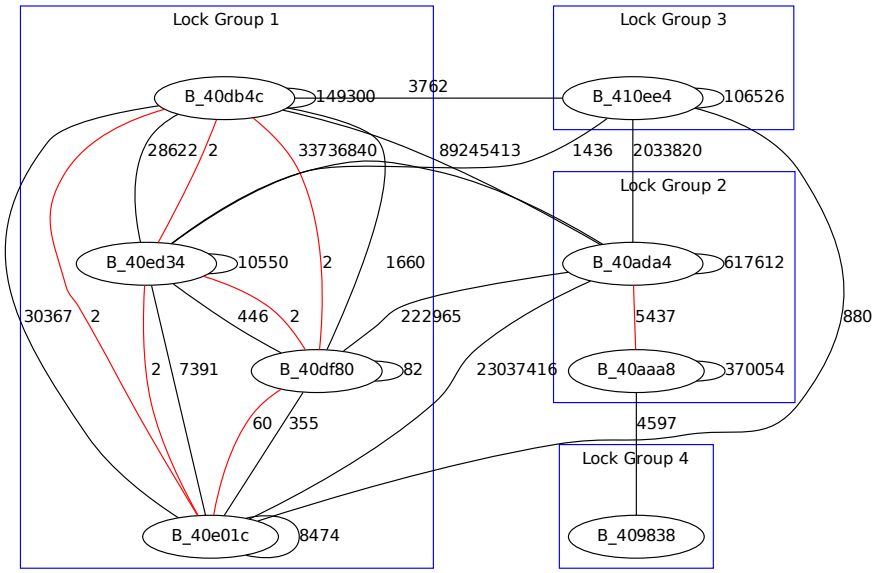


Figure 39. Example of generated graphs with sharing edges (red) and interaction edges (black).

We devised an algorithm that aims to strike a balance between too many lock instances and too few. We start off by assigning one potential lock for each shared data word. Then we reduce the number of locks required by merging words into groups that are often are accessed together. A detailed pseudocode that implements this grouping is shown below. We first collect the number of times any pair of words are accessed at the same critical section instance (word_rel), and afterwards calculate the probability when one word is accessed, the other will, too (relative_probability).

Listing 1. Pseudocode that implements the per-object locks

```
perobject_locks(tracefile, lock_instance, critical_sections):
    foreach critical section in critical_sections
    foreach dynamic instance of the critical section dyn_cs:
        add all write addresses to write_set(dyn_cs)
        foreach pair of words in write_set(dyn_cs):
            increment word_rel[w1][w2]
            increment count[w1], count[w2]

    foreach dynamic instance of the critical section dyn_cs:
        foreach pair of words in write_set(dyn_cs)
            relative_probability = word_rel[w1][w2] / (count[w1] + count[w2])
            if relative_probability > threshold:
                group both words together

    OUT: Each group of words can be protected by distinct locks
```

In Figure 40, we have part of the word relationship graph that was generated by the dynamic lock splitting algorithm run on radiosity from SPLASH-2. For each node, which represents a single word, the edges to other nodes have weights that represent the relative probability of them being accessed together (the absolute number of times each pair was seen together is in parentheses). We can see that there is a strong correlation between some words. As a result, we ca strike a balance between one static lock protecting all words and one lock per word - instead, we can have one lock per group of words.

Looking back at the results for radiosity, for 96 words we are down to 32 lock instances, which leads to almost perfect scalability up to 64 cores, as can be seen in the following figure (Figure 41).

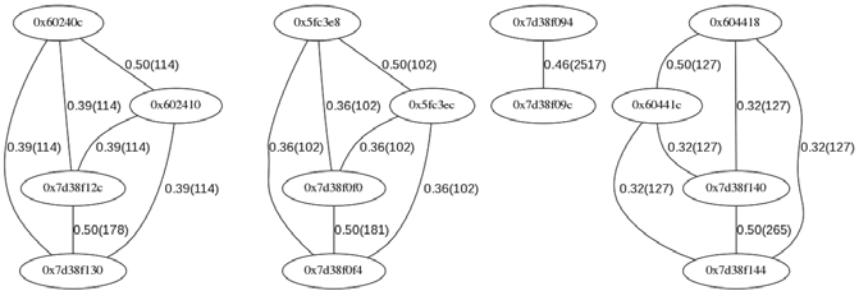


Figure 40. Word relationship graph for dynamic lock splitting algorithm.

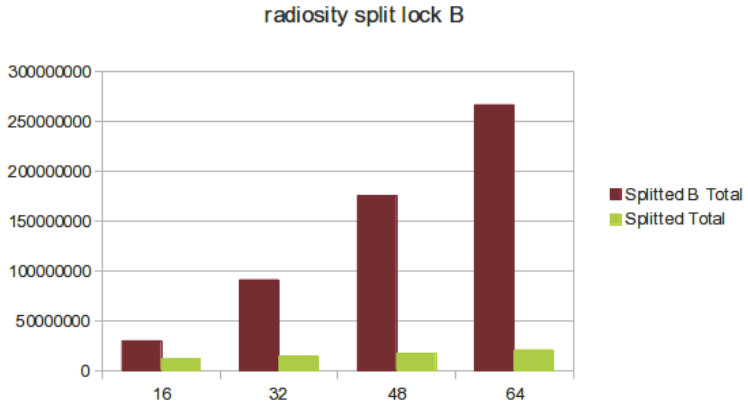


Figure 41. Dynamic lock splitting result for radiosity.

4.3 Integration of Tools for Synchronization-Related Limiters

Waiting during synchronization operations is usually easy for a tool to recognize and account for, and in most cases such accounting can be done without perturbing the application's execution much. Recognition and accounting for of synchronization tends to be easy because the application usually relies on well-defined synchronization operations provided by the runtime library, allowing a profiling tool to detect synchronization operations at runtime simply by intercepting these library functions. An exception to this is "hand-made" synchronization, e.g. when a program uses an ordinary variable for synchronization without calling library functions. However, hand-made synchronization tends to be poorly portable across platforms (what works under one consistency model might not work under another) and mostly consists of spin-waiting until the value of a variable changes, a coding pattern that is typically easy to recognize and intercept by a profiling tool's program analysis and instrumentation.

We have built a research tool that performs this interception and accounting of synchronization waiting time for both barriers (LIME) and locks. The tool accounts separately for synchronization waiting (a threads has to stop and wait) and overheads (a threads spends time to complete a synchronization operation even when waiting is not needed), and provides attribution of waiting and overheads to both synchronization variables (how much waiting and/or overhead was spent on each synchronization variables) and code point (how much waiting and/or overhead was spent by a particular line of code that calls a synchronization operation). This level of detail was needed to understand the problem better and to provide input into our load imbalance and lock contention analysis and reporting tools.

We use comparable/additive metrics for how much load imbalance and how much lock contention exists in a given application run. In both cases, the metric we use is time spent waiting divided by execution time. This makes it easy to determine which of the two synchronization limiters are present (non-negligible waiting time on that type of synchronization) and which one is a more pressing problem for the programmers to address first (the one with a larger waiting time).

5 Educational Activities and Results

5.1 Motivation

Many-core processors can only approach their performance potential if software is written to exploit a large number of cores. The long-term trend is now toward incrasing the core counts rapidly into the foreseeable future, and the long-term viability of this trend relies on software being written in a way that allows it to scale to increasing numbers of cores. Unfortunately, most programmers can achieve signficant speedups (relative to sequential execution) with a small number of cores after a prolonged effort, and true performance scaling to

many cores is a major challenge or even beyond their reach. One of the possible reasons for this is that programmers may not be familiar with typical performance scaling limiters, and in some cases they may not be aware of a particular limiter even as a concept.

The main goal in this NSF/SRC MCDA project is to identify software and hardware mechanisms that help programmers identify performance scaling limiters present in their code and, subsequently, change their code to make it scale better. In addition to general uses of such tools in software development, the project has investigated performance scaling in the educational setting, especially in terms of understanding what would be needed to help students learn how to develop scalable parallel code.

Three types of tools and mechanisms were designed in this project - those that report scaling limiters related to excessive cache misses, thsoe that identify the reasons for load imbalance in the code, and those that identify lock contention and offer suggestions on how to reduce this contention. Over the past decade, the PI has taught several classes that had parallel programming assignments targetting >16 cores. The questions students ask and programming issues they come to ask about during office hours, it is obvious that many do not clearly understand the relationship between performance scaling bottlenecks and performance, or how different types of bottlenecks affect each other. In particular, students often approach scaling problems through a trial-and-error approach - they have no concrete ideas about what the real problem might be, so they go through lecture slides that explain some ofthe bottlenecks, and attempt code changes similar to those shown in lecture slides. This involves a lot of unneccessary programming effort - e.g. students might go through exraordinary amounts of effort to improve load balance in their code, altohugh the code actually does not have much load imbalance. Worse, this approach of identifying the problem by checking to see which "fixes" work can lead students to disregard the real problem because the "fix" for it didn't help, e.g. because the "fix" was implemented poorly, introduced another bottleneck, etc.

5.2 Curricular Constraints

The natural venue for teaching scalability limiters are computer architecture classes, where students are taught cache organization and cache coherence mechanism, and are thus prepared to learn about the cache misses that result from these mechanisms, and where they learn about synchronization primitives and the resulting load balancing, lock contention, and other issues. An example of such classes at Georgia Tech is CS 2200 "Computer Systems and Networking", where undergraduate students are briefly introduced to these necessary concepts, and CS 4290 "Advanced Computer Organization" (undergraduate) and CS 6290 "High-Performance Computer Architecture" (graduate, co-taught with CS 4290), where both undergraduate and graduate students are taught the details of the cache coherence and how synchronization is implemented.

The ideal class for this would be CS 2200 - most CS undergraduates take this class, so introduction of these concepts and skills in CS 2200 would have the broadest impact on the skill-set of an average graduating student. Ideally, every CS graduate that expects to do any multi-core programming in their future career would understand this important scaling limiter and be intellectually equipped to tackle it. The disadvantages of teaching cache-miss performance debugging mostly revolve around class time - this class is a very fast-paced introduction to many key CS concepts, providing a review of important topics from computer architecture, operating systems, and networking. As a result, the total amount of time spent on caches (from introducing the concept of caches to students who have not been exposed to that concept yet, to showing an example of how cache coherence works) is usually only 4-6 lecture hours. The time to explain cache-miss performance debugging in this context would only be about 15-30 minutes. The PI has taught CS 2200 several times in the past and attempted to cover this topic in this amount of time - the result is that only a few students remember the concepts a few weeks later (in a pop-quiz that asks very basic questions about coherence misses), and many do not even remember that a concept called "true sharing" exists.

The next-best class for teaching cache-miss performance debugging is CS 4290/6290, which is a "pick" class for students that take the "platforms" thread through the CS undergraduate program. A thread acts much like a specialization, and "platforms" is a thread that focuses on hardware and systems issues. This reduces the potential impact of introducing cache-miss performance debugging - many students graduate by taking other threads and still get jobs in which they need to do significant multi-threaded programming. However, the class devotes significant time (1-2 lecture hours) specifically to types of cache misses (including true and false sharing). In the PI's experience, this is sufficient for most students to internalize these concepts sufficiently for recognition, but not enough to gain any expertise or skills in how to alleviate them - that would be best accomplished through a project assignment. At the very beginning of this NSF/SRC project (Fall 2009 semester), the PI has introduced such an assignment with mixed results, as will be explained next.

5.3 The Cache-Miss Performance Debugging Assignment

The assignment asked student to parallelize the "game-of-life" algorithm, where "cells" are represented as elements of an array and "live" or "die" in each time-step according to how many live neighbors they have. They were also given a naive parallel implementation - cells are mapped round-robin to the available threads, and in each time-step a thread updates the cells mapped to it and then synchronizes on a barrier with other threads. In this naive implementation the false sharing occurs on nearly every memory access to a cell, thus resulting in parallel performance that is (significantly) worse than a single-threaded implementation.

The students also had access to a rudimentary simulation-based tool that classifies cache misses into the appropriate categories (capacity, coherence, true sharing, false sharing). This tool was a direct result of the PI's NSF/SRC-funded research on cache miss classification - this research and the cache miss classification algorithm were described in a recent research paper that has been accepted for publication in ACM Transactions on Architecture and Code Optimization.

The results were mixed for two reasons. First, many students chose not to use the miss-classification tool - since the tool is simulation-based, it executes the student's programs at a rate of only a few million instructions per second, i.e. a factor-or-1000 slowdown relative to native execution. This made the use of the tool non-interactive and frustrating for these students, and they decided to try and understand the cache-miss problems by trial-and-error. Second, many students simply gave up after a few simple optimizations that resulted in some parallel

speedup but very poor scaling - most students in this category had speedups of only about 2X even when using 64 cores. Part of the reason for this could be that students needed to run simulations to get performance results for more than 4 cores - due to resource constraints, no real hardware with more than 4 cores was made available to students at the time.

The assignment was part of a class on computer architecture, so 64-core performance was measured using a cycle-accurate simulator because no 64-core machines were available at the time. The assignment was due 20 days after it was released, and yet 36% of the programs submitted by students either did not work at all, or achieved parallel speedups of 4x or less on in 64-core execution. In 6% of the submissions, parallel speedup was between 4x and 8x, in 3% of submissions it was between 8x and 16x, in 11% of submissions it was between 16x and 24x, in 36% of submissions it was between 24x and 32x, and only 8% of submissions it was above 32x. From these results and from interactions with students, it appears that 36% of the students never got beyond just trying to get the parallel code to work. They were given the serial code when the assignment was released, along with a skeleton parallel version (that creates and joins threads, shows how to use locks and barriers, etc.), so most of the difficulty was in actually parallelizing the code and not in how to use these pthreads constructs. Relatively few students achieved scaling higher than 4x but lower than 24x, which suggests that there is a steep learning curve between virtually no scalng and reasonable scaling - once students learn how to improve scaling beyond just a few cores, they usually succeed in improving it significantly. In this case, speedup of about 24x or better required the students to avoid serialization on locks and to avoid excessive false sharing of array elements. Many initially did all each array update in a critical section, either with a single lock for the entire array or with a separate lock for each element, and both approaches lead to slowdowns relative to serial execution due to lock overheads and serialization (with one lock). Once excessive locking was eliminated, the speedup would be around 6x. False sharing was introduced if students partitioned array elements to threads in particular ways - many chose to assign individual elements round-robin, resulting in a huge increase in false sharing misses and limiting speedups to only 1-3x. Some assigned rows or columns round-robin, leading to significant false sharing and limiting speedups to 8-20x, and most of those that achieved $>$24x speedups divided the array into blocks. It should be noted here that students were warned about false sharing and given an initial implementation of our coherence miss classification mechanism (i.e. it reports how many false sharing and how many true sharing misses are encountered at each static load/store instruction, which can easily be projected back to the source code using the addr2line tool from the GCC compiler framework), so relatively few students got past he excessive-locking problem and chose not to do something about false sharing. With this in mind, it is somewhat alarming that more than 10% of the students still had false sharing as the dominant scaling limiter.

However, the results of this assignment were also encouraging for in two ways. First, several students managed to get near-linear or even super-linear speedups even for (simulated) 64-core execution. This is encouraging because it shows that a significant fraction of the students can understand and relatively rapidly solve lock-related and cache-miss-related performance problems. Second, it appears that most of the students who achieved relatively good scalability did so by using the cache miss classification tool - many by finding ways to tolerate the excessive simulation time, e.g. by doing other work while waiting for simulations to finish. This seems to suggest that better tools (e.g. hardware support for cache miss classification, as proposed by PI and his collaborators in their recent research publications) can be expected to improve the cache-miss performance debugging process by both 1) making the process less frustrating to those who would have used simulation-based tools to get good results and 2) encourage others (who chose not to use the simulation-based tool) to use the tools and possibly get better results.

5.4 Future Plans and Curricular Constraints

The problem of simulation-induced slowdowns (and the resulting frustratingly-slow program execution when the tool is used) can be somewhat alleviated by omitting some details in the simulation - most of the simulation slowdown, at least in SESC, the simulator that the PI and his students used, is caused by detailed modeling of out-of-order execution in each core. By using simpler (e.g. in-order, or even one-instruction per cycle) core models together with more detailed cache simulation (to accurately classify cache misses and get reasonable estimates of their performance impact), the simulation can be sped up by almost an order of magnitude. This might be sufficient to at least somewhat overcome the problem of giving up on the tool. Additionally, the PI plans to more strongly encourage students to use the tools, at least to diagnose the remaining problems after a potential solution is developed. In the Fall 2009 class, the PI has allowed the use of the tool but has neither encouraged nor discouraged its use - at the time the PI did not know whether the tool would actually be beneficial, compared to simply trying to figure things out without the tool, so he felt that it would not be fair to students to lead them to use a potentially not-so-useful tool. Now that there is at least some evidence of the tool's benefits, the PI can encourage (but still not require) the students to use the tool.

Unfortunately, teaching assignments in CS at Georgia Tech, like in many other schools, are subject to many constraints in addition to the PI's wishes. As a result, the PI has not had an opportunity to teach neither CS 2200 not CS 4290/6290 since Fall 2009. The PI is scheduled to teach CS 4290/6290 in Spring 2013, after the expiration of this project. At that time, the cache miss, load imbalance, and lock contention/overhead tools will all be available for use by students.

6 Conclusions and Current/Future Research Directions

6.1 Publications

The cache miss classification framework was published in Guru Venkataramani's PhD thesis, and Section 2 largely mirrors the contents of that thesis. The coherence miss classification component of this work was recently also published in ACM Transactions on Architecture and Code Optimization (TACO).

The LIME tool for load imbalance analysis and reporting was presented in 2011 at the ACM International conference on Software Engineering (ICSE), a top software engineering conference.

Finally, our lock contention analysis and reporting tool is still work in progress, and we are planning to submit a research paper based on this work to one of the top conferences with upcoming paper submission deadlines: HPCA, ASPLOS, or ISCA.

6.2 Conclusions and Ongoing Work

The tools presented in this report address several major problems in achieving good performance on multi-core processors: excessive cache misses, load imbalance, and lock contention. In all three cases, the existence of the problem is relatively easy to detect, and the challenge lies in how the problem can be reported to programmers in an actionable way, i.e. in a way that suggests which changes in the code can help. Our cache miss classification addresses this problem by identifying the type of misses that are the most detrimental to the application's performance, so the programmers can use the cache miss reduction approaches suitable for that type: pad shared data to avoid false sharing, reduce communication among threads to avoid true sharing, change the layout or alignment of variables to avoid conflict misses, or improve memory access locality to avoid capacity misses. Our LIME load-imbalance tool provides actionable reporting by pinpointing the line of code where load imbalance is introduced, which leads the programmers directly to what should be done to improve load balance. Finally, our lock contention tool identifies which critical sections may be amenable to finer-grain locking and suggests to the programmer the finer-grain locking approach that is expected to provide the most performance benefit with as little programmer effort as possible.

6.3 Integration of Tools for Cache Miss and Synchronization Limiters

The profiling for all three of our main tools (cache miss, load imbalance, and lock contention) can be used together in a single simulation run, and each tool can then be used to report the results relevant for that limiter. As indicated in Section 4.3, we can also point the programmer in the right direction in terms of which synchronization-related limiter (if any) to try to alleviate. The situation is more difficult for cache misses - while we can report which misses are the dominant ones and what is causing them, we do not have a good way of determining how much of a problem cache misses are in a particular run and how much performance benefit can be expected from fixing them. Our simulator has the ability to attribute stall time (and thus execution time) to cache misses at each level of the hierarchy, but further work is needed to isolate cache misses associated with synchronization waiting. Also, more work is needed to identify interactions between the three types of limiters, e.g. identify the impact that an improvement in load balance would have on lock contention and on miss rates, to identify the impact that lower lock contention would have on load imbalance and cache misses, and to identify the impact that lower miss rates at certain points in the code would have on lock contention and load imbalance.

6.4 Future Research Directions

Our future work in this domain includes further refinement of the reporting tools, e.g. performance projection for potential cache miss reduction approaches, reporting of contention-based limiters (such as unfair bus or cache contention) in LIME, and performance projection for lock -splitting approaches. We are also very interested in pursuing integration of our approaches into production software development tools and environments, such as Intel Parallel Advisor or IBM's Parallel Environment, Eclipse, etc.

In our lock contention work, another very interesting direction is to create a framework that accurately reports on the tradeoff between locking overheads and lock contention - finer-grain locking can result in more cache pressure (more lock variables) and nested critical sections (when using variables protected by different fine-grain locks), so the best solution performance-wise typically involves very judicious use of fine-grain locking to maximize contention reduction without introducing a lot of new overhead.

Unfortunately, after the expiration of this NSF award and the corresponding SRC contract, the PI has no remaining funding for this line of work, so he is planning to apply for new funding to both NSF and SRC over the next year.

7 *

References

[1] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *9th Symp. on Operating Systems Principles (SOSP)*, 2003.

[2] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev. Parallelization Made Easier with Intel Performance-Tuning Utility. *Intel Technology Journal*, Nov. 2007.

[3] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *36th Int'l. Symp. on Computer Architecture (ISCA)*, 2009.

[4] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *2008 Int'l. Symp. on Workload Characterization*, 2008.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13(7):422426*, July 1970.

[7] M. Brorsson. SM-prof: a tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 178–187, New York, NY, USA, June 1995. ACM.

[8] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *17th Int'l Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 2008.

[9] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149, 1998.

[10] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. *SIGPLAN Not.*, 34(5):1–12, 1999.

[11] J. D. Collins and D. M. Tullsen. Hardware Identification of Cache Conflict Misses. In *Proceedings of the International Symposium on Microarchitecture*, November 1999.

[12] G. Corder and D. Foreman. *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. Wiley and Sons, Inc., 2009.

[13] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. ProfileMe : Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[14] P. Dubey. Recognition, Mining and Synthesis moves computers to the era of Tera. In *Technology@Intel Magizine*, Feb. 2005.

[15] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *ISCA*, pages 88–97, 1993.

[16] M. Efroymson. Multiple regression analysis. In *Mathematical Methods for Digital Computers*, pages 191–203. Wiley, 1960.

[17] S. Eggers and T. Jeremiassen. Eliminating false sharing. In *International Conference on Parallel Processing*, pages 377–381, Aug. 1991.

[18] GCC Team. GCC Toolchain. *http://gcc.gnu.org*, 2009.

[19] G. H. Golub and C. F. Van Loan. *Matrix computations*. John Hopkins Univ. Press, 1996.

[20] F. Hao, M. Kodialam, and T. V. Lakshman. Building high accuracy Bloom filters using partitioned hashing. *SIGMETRICS Perform. Eval. Rev.*, 35(1):277–288, 2007.

[21] J. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan-Kaufmann Publishers, Inc., 2nd edition, 1996.

[22] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, EECS Department, University of California, Berkeley, Nov 1987.

[23] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence Decoupling: Making use of incoherence. In *ASPLOS*, pages 97–106, 2004.

[24] Hypertransport Consortium. Hypertransport technology-1.0. *http://www.hypertransport.org/*, 2009.

[25] Intel Corp. Intel threading building blocks 3.0, 2010. `http://www.intel.com/software/products/tbb/`.

[26] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, 2002.

[27] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[28] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. In *12th Int'l. Symp. on High Performance Computer Architecture (HPCA)*, 2006.

[29] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comput.*, 55(6):769–782, 2006.

[30] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991.

[31] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *34th Int'l. Symp. on Computer Architecture (ISCA)*, 2007.

[32] R. Larsen and M. Marx. *An Introduction to Mathematical Statistics and Its Applications*. Pearson, 2000.

[33] K. Lepak and M. Lipasti. Temporally silent writes. In *Architectural Support for Programming Languages and Operating Systems*, 2002.

[34] J. Li, J. F. Martinez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *10th Int'l. Symp. on High Perf. Computer Architecture (HPCA)*, 2004.

[35] R. G. Lomax. *Statistical Concepts: A Second Course*. Lawrence Erlbaum Associates, 2007.

[36] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[37] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[38] H. Mousa and C. Krintz. HPS: Hybrid Profiling Support. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 38–50, Washington, DC, USA, 2005. IEEE Computer Society.

[39] P. Nagpurkar, H. Mousa, C. Krintz, and T. Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim.*, 3(1):35–66, 2006.

[40] M. J. Norusis. *PASW Statistics 18 Statistical Procedures Companion*. Pearson, 2010.

[41] OpenMP Architecture Review Board. Openmp application program interface. Technical report, 2010. `http://openmp.org/wp/openmp-specifications/`.

[42] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *34th Int'l. Symp. on Computer Architecture (ISCA)*, 2007.

[43] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[44] C. Sanderson. Armadillo library, 2010. `http://mloss.org/software/view/176/`.

[45] S. S. Sastry, R. Bodík, and J. E. Smith. Rapid profiling via stratified sampling. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 278–289, New York, NY, USA, 2001. ACM Press.

[46] T.-P. Shi and E. S. Davidson. Grouping array layouts to reduce communication and improve locality of parallel programs. In *International Conference on Parallel and Distributed Systems*, 1994.

[47] Sun Microsystems. UltraSPARC T2 Supplement. *UltraSPARC Architecture*, 2007.

[48] N. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *14th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[49] N. Tallent, J. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *15th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[50] J. Torrellas, M. J. Lam, and J. L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache misses. In *Proceedings of the International Conference on Parallel Processing*, pages 266–270, 1990.

[51] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic. Coherence Miss Classification For Performance Debugging in Multi-Core Processors. In *Thirteenth Workshop on Interaction between Compilers and Computer Architecture*, Los Alamitos, CA, USA, 2009. IEEE Computer Society Press.

[52] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture*, 1995.

[53] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[54] C. B. Zilles and G. S. Sohi. A programmable co-processor for profiling. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 241, Washington, DC, USA, 2001. IEEE Computer Society.

NSF Award 0903470, “Performance Debugging Support for Many-Core Processors”

Final Report

Activities

In the first year of the project, we have performed the following set of activities:

1) Proposed a new, programmer-friendly, definition of what is a false sharing miss, performed a design space exploration for possible approximations of this definition for use in on-line performance counting in future processors, and submitted this work for possible publication in the ACM Transactions on Architecture and Compiler Optimization (TACO) journal.
2) Devised a new automated methodology for more precisely identifying the cause of load imbalance in parallel programs and reporting this cause to programmers. We expect to submit this work for publication in the next few months.
3) Performed an initial qualitative analysis of performance limiters in parallel code written by undergraduate and graduate students, with the goal of both a) trying to identify the key concepts that must be taught to students to improve their ability to effectively solve performance-related problems in parallel code, and 2) identifying the common types of limiters so they can be targeted by tools and techniques we will develop in our research work.

In addition to these research and education activities, the PI has presented the results of our work in the SRC project kick-off meeting and, more recently, at the SRC Annual Contract Review meeting. These presentations have resulted in additional meetings with researchers from Freescale, who have expressed interest in potential applications of our performance debugging methodologies in embedded systems.

In the second year of the project, we have performed the following set of activities:

1) Revised the programmer-friendly definition of true and false sharing misses and the corresponding design space exploration of approximations for use in on-line performance counting in future processors, which has been accepted for publication in the ACM Transactions on Architecture and Compiler Optimization (TACO) journal.
2) Implemented a new automated methodology for pinpointing the cause of load imbalance in parallel programs and reporting this cause to programmers. The portion of this work that deals with work imbalance in the code will be presented in the International Conference on Software Engineering (ICSE, this is the top conference in software engineering) in May 2011, and the part on hardware-related causes of imbalance is being prepared for a separate publication.
3) Performed initial examination of causes of lock contention and identified promising approaches to pinpointing and reporting performance limiters related to lock contention.

In addition to these research and education activities, the PI has presented the results of our work in the SRC Annual Contract Review meeting, and has continued interactions with researchers from Intel and Freescale on making this work relevant to real-world problems.

In the third (final) year of the project, we have performed the following set of activities:

1) Developed a simulator-based and pin-based profiling frameworks for reporting lock contention (lock waiting time) in a programmer-friendly way.
2) Developed an automated tool for recommending targeted use of fine-grain locks, along with expected performance improvement from each change.

In addition to these research and education activities, the PI has presented the results of our work in the SRC Annual Contract Review meeting, has continued interactions with researchers from Intel and Freescale on making this work relevant to real-world problems, and has presented this work during visits to two Intel sites (Hudson/Boston and Hillsboro/Portland) and to AMD.